2008

# Feature-based calibration of distributed smart stereo camera networks

Aaron Mavrinac
*University of Windsor*

# FEATURE-BASED CALIBRATION OF DISTRIBUTED SMART STEREO CAMERA NETWORKS

by

**AARON MAVRINAC**

A Thesis
Submitted to the Faculty of Graduate Studies
through Electrical and Computer Engineering
in Partial Fulfillment of the Requirements for
the Degree of Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada
2008

*Your file Votre référence*
*ISBN: 978-0-494-42305-9*
*Our file Notre référence*
*ISBN: 978-0-494-42305-9*

# Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

# Abstract

A distributed smart camera network is a collective of vision-capable devices with enough processing power to execute algorithms for collaborative vision tasks. A true 3D sensing network applies to a broad range of applications, and local stereo vision capabilities at each node offer the potential for a particularly robust implementation. A novel spatial calibration method for such a network is presented, which obtains pose estimates suitable for collaborative 3D vision in a distributed fashion using two stages of registration on robust 3D features. The method is first described in a general, modular sense, assuming some ideal vision and registration algorithms. Then, existing algorithms are selected for a practical implementation. The method is designed independently of networking details, making only a few basic assumptions about the underlying network's capabilities. Experiments using both software simulations and physical devices are designed and executed to demonstrate performance.

# Dedication

This work is dedicated to Crystal, my best friend.

# Acknowledgements

First and foremost, I thank my thesis advisors Dr. Xiang Chen and Dr. Kemal Tepe, for their invaluable guidance and support throughout this research work. I also thank Dr. Mohammed Khalid, my departmental reader, as well as Dr. Arunita Jaekel, my external reader, for their own advice toward the completion of my research and this thesis. I appreciate the instruction and guidance offered by Dr. Maher Sid-Ahmed, Dr. Jonathan Wu, Dr. Huapeng Wu, and Dr. Bubaker Boufama during my studies.

I am indebted to my fellow students for their support and frequent assistance; in particular, I would like to thank Mr. Ahmad Shawky for his mentoring in the theory and practice of computer vision early on and his continued support and advice throughout my research.

My thanks go out to various other faculty and staff at the University of Windsor who in one way or another supported my work. The department technicians, Mr. Don Tersigni and Mr. Frank Cicchello, and the Technical Support Centre staff provided materials and expertise instrumental to the experiments herein.

My deepest gratitude goes out to members of the academic community and of free software projects worldwide, who put in countless hours of hard work and, believing in the spirit of community, make that work freely available on the Internet. The innumerable software packages, code snippets, and reference pages I used every day literally made this work possible.

Finally, I would like to acknowledge the financial support provided by the Natural Sciences and Engineering Research Council of Canada throughout my research.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Distributed Smart Cameras

The field of *distributed smart cameras* is one still in its infancy, largely unexplored by the literature purely on its own. It combines research from a wide variety of fields, including computer vision, image and signal processing, embedded systems, distributed systems, and communications. As with any other emerging area of research with a diverse lineage, there is an inherent challenge in defining its scope in a way that embraces all the possibilities offered by this new technology. The challenge is twofold: as research from the various parent fields converges on the nascent one, it must both account for new limitations imposed by the intersection of the fields and see new opportunities presented by their union.

*Visual sensor networks* are a concept closely related to distributed smart cameras. The two overlap almost fully, particularly in the extant body of research, and indeed they may soon converge into a single area of study. However, at present, it is important to keep their definitions as broad as possible so as not to exclude potential opportunities, and so it would not be entirely accurate to use the terms interchangeably, nor to call either a subset of the other. To illustrate, one might envision a visual sensor network not composed of smart camera devices, or a distributed smart camera system which does not embody the fundamental concept of a sensor network.

The type of system envisioned in this research falls fully within the intersection of the two fields: it could accurately be called either a distributed smart camera network or a visual sensor network. The emphasis, however, is on the distributed nature of the algorithms used, and in that light, a definition is here provided for the former term which shall be used throughout this thesis:

> A *distributed smart camera network* is a group of physically dispersed devices, each capable of sensing and locally processing visual data from the environment via one or more digital cameras, which may communicate with one another and/or with other devices to perform collective processing of this data.

This definition is quite general, so to further delimit the type of system this research pertains to, a set of

paradigms are applied to it, outlined and described in Section 1.2.

## 1.2 Paradigms

### 1.2.1 Homogeneous Distributed Network

The definition of a distributed smart camera network in Section 1.1 may, in fact, be *too* general. The name itself implies that collective processing of the data throughout the network should be *distributed* in nature, not simply that the devices be physically distributed over some geographic area; that is, the nodes themselves should use some of their local processing power to collaborate in global processing over the network, rather than just forwarding their local results to a central station.

This is a highly advantageous system design which lends itself extremely well to this type of system. Smart camera nodes gather an enormous amount of information compared to other types of sensor nodes (which typically measure only one or a few values), and though they distill this information with local processing to a degree that varies by application, processing the information collectively is in all practical cases a computationally intensive task. Fortunately, the type of processing envisioned includes tasks like tracking, registration, reconstruction, and other highly parallelizable algorithms. Furthermore, the data which need to be processed together tend to be localized geographically in large networks, so that the associated communication is also compartmentalized in networks relying on physical proximity. Thus, the network becomes extremely scalable, as each node added to the network also adds distributed computation power, which, aside from a relatively small amount of overhead and inefficiency in parallelization, makes up for the load it adds.

As with scalability in processing, distributed data yields an opportunity to design extremely reliable and robust storage and retrieval systems. In whatever final form it takes after processing, visual data about the same space-time area might be stored redundantly on multiple nodes, so that if some nodes fail others are fully equipped to respond to queries.

In addition to run-time scalability and reliability, the distributed paradigm offers one additional major advantage. If the network is a homogeneous one – one in which all nodes are physically identical and run the same algorithms with the same initial conditions – the complexity and cost of development, manufacturing, and deployment are significantly reduced, since only one type of node need be produced and installed and the network initializes itself in an ad-hoc manner.

### 1.2.2 True 3D Sensing

The bulk of research in distributed smart cameras to date has centered on systems involving single cameras at each node. The focus is usually on specific tracking and surveillance applications which allow for some simplifying assumptions about the contents of the scene, the relative locations of the nodes, etc., so various 2D computer vision algorithms are applied within these constraints to solve such problems.

A true 3D sensing network, as advocated in [1], can be applied to a more general class of problems. For instance, the tracking problem is greatly simplified once 3D positions of the detected objects are known.

*Virtual views* – reconstructions of the scene from arbitrary viewpoints in space and time, possibly at greater resolution than a single node can provide – would be immensely useful to surveillance, manufacturing, and many other applications. But in addition to improving existing applications, whole new ones can be envisioned that may not be possible at all without full 3D sensing.

### 1.2.3 Stereo Camera Nodes

While it is possible to create a 3D sensing network using single-camera nodes, this presents a series of challenges. In general, it is assumed that a node does not initially know its location and orientation, or *pose*, relative to the other nodes in the network. As will be seen in Chapter 2, some work has been done in calibrating networks of such nodes for 3D vision, but without the constraint of a short baseline, this is an exceptionally difficult problem. Apart from this, even if the nodes are perfectly calibrated, 2D point correspondences for 3D triangulation are still very difficult to find due to occlusion and variations in the reflective properties of a surface.

These problems are minimized if the stereo baseline is much shorter than the distance to the scene. Therefore, if stereo cameras are used at each node, 3D features can be recovered locally, and then any network-level processing operates purely on robust 3D data. The calibration problem in 3D then becomes a *registration* problem, where various iterative statistical methods can be employed to yield a much more accurate pose estimate than, say, the bilinear tensor (fundamental matrix) in the wide-baseline stereo case provides. Using such nodes relaxes the complicated deployment and scene constraints imposed by single-camera nodes, and provides more complete and accurate 3D reconstruction capabilities overall.

## 1.3 Thesis

The focus of this thesis is on the first step in building any functional distributed smart camera network: calibration. In order to perform any sort of useful collaborative processing of visual data, it must be known how one piece of data relates to another.

If the totality of what a distributed smart camera network observes is thought of as existing within a space-time coordinate system, as in [1], it is required that each node be able to reconcile its own locally observed data with a world reference frame, and thereby to the data from the other nodes. The world reference frame need not be any particular coordinate basis; indeed, it may in fact be an abstract conceptualization of what is, in reality, simply a series of pairwise relative mappings between nodes. The important factor is that each node has or can obtain the knowledge necessary to align its own data.

This thesis concentrates on the spatial portion of this calibration. The assumptions in Section 4.2 reduce the problem to development of a distributed algorithm for geometric localization and orientation (Chapter 4) and the selection of appropriate computer vision methods for practical use (Chapter 5). Temporal synchronization is implicitly satisfied by these assumptions, but for future work, there do exist means of achieving it explicitly in a distributed network [55, 62]. The assumptions also abstract away most of the networking details, so that the algorithm does not depend on any particular topology, medium, or protocol.

### 1.3.1 Statement

The objective of this thesis is to develop a distributed spatial calibration algorithm capable of localizing and orienting the coordinate systems of a series of homogeneous smart stereo camera nodes within a world coordinate system, solely based on the three-dimensional visual data obtained by the nodes, where the nodes have no *a priori* knowledge of their location and orientation within the environment or with respect to one another. The algorithm should not require any manual input whatsoever, nor should it inherently depend on any particular configuration of the scene (although, as will be discussed, some general conditions might be necessary in practical implementations). To achieve this objective practically, in addition to implementing the algorithm itself, appropriate underlying computer vision methods shall be selected from the 2D images up. The method developed shall be independent of the topology, medium, and protocols used to network the smart camera nodes, aside from some general assumptions about the network's capabilities.

### 1.3.2 Methodology

With the objective and paradigms established, Chapter 2 provides a survey of the literature justifying the concepts and the opportunity for improvement, and covers the state of the art in the two major relevant computer vision topics. Chapter 3 covers some basic theory and conventions, providing a framework within which the remainder of the work is developed.

The calibration problem itself is approached in two stages. First, in Chapter 4, the calibration problem is examined in purely geometrical terms, as stated in Section 4.1. A series of local assumptions are made (Section 4.2.6) to temporarily satisfy some practical requirements abstractly, so that the algorithm, described in a generic way in Section 4.4, concentrates wholly on realizing a distributed solution to the geometrical problem. Second, in Chapter 5, the practical requirements of the algorithm are examined, and the local assumptions used in the previous chapter are replaced by suitable computer vision methods.

Chapter 6 describes several simulated and physical experiments, and reports on their results. It outlines the performance metrics (Section 6.1) used to evaluate the algorithm's performance, and the equipment and software (Section 6.2) used to perform the experiments. Three types of experiments are described along with their results, followed by a comparison with existing calibration algorithms.

# Chapter 2

# Literature Survey

## 2.1 Distributed Smart Camera Networks

### 2.1.1 Justification of the DSSC Concept

The concept of distributed smart stereo cameras brings together three distinct advantageous paradigms:

1. Passive 3D (Stereo) Vision
2. Multiple Views
3. Distributed Collaborative Processing

These have been been used extensively to improve on the basic operations of computer vision over what is possible with traditional monocular methods. The underlying premise of distributed smart stereo camera network research is that combining all three will allow for yet better low-level solutions and a new range of high-level applications.

This section explores how in the literature the above paradigms have been used, albeit not all concurrently, to enhance some of the most important primitive vision operations. The general benefits and limitations in each case are summarized, and then the advantages of employing all three paradigms in the form of a distributed smart stereo camera network are extrapolated.

#### Shape Recognition

The 3D shape information offered by stereo vision is richer and much more robust than 2D contours in the monocular case. It can be similarly reduced to metrics for comparison to a database, as in [36, 37], greatly improving recognition performance and extending the range of applicability. However, such methods still suffer from the fact that a full 3D reconstruction is not available from a single view, and without application-specific constraints, this can only be mitigated by cumbersome schemes of moving the target relative to the camera rig or by complicated machine learning methods to account for the incompleteness of the information.

Distributed smart stereo cameras, in contrast, offer the potential for a full 3D reconstruction of the object without constraints, allowing recongition methods to yield more robust performance without added complexity.

### Object Tracking

Both stereo vision and the use of multiple views allow objects to be tracked in three dimensions, which is generally not possible in monocular vision. Stereo vision has been posited for tracking [38], since it offers direct 3D information, but its usefulness is limited by the fact that objects can only be tracked within the field of view of a single camera rig. Much research has been done, particularly for surveillance and environment monitoring applications, into tracking across multiple views, both overlapping [39, 40, 42] and disjoint [41]. More recently, the advantages of a decentralized (distributed) approach to multi-view tracking using smart camera devices have been explored [43, 44, 45].

The complexity of decentralized tracking across multiple views could certainly be reduced if the analysis were performed on explicit 3D information rather than interpretation of 2D images; distributed smart stereo cameras are thus in a position to greatly improve object tracking in the general case.

### Motion Analysis

Motion analysis, the extraction of quantitative data from dynamic scenes, is another area which has benefitted from both stereo vision and multiple views. It is possible to recover richer 3D motion data from stereo image sequences [46, 61, 47] than in the monocular case, allowing for more informative high-level analyses. Motion estimation from multiple views has been heavily researched [48, 49], particularly due to its obvious applications in surveillance and environment monitoring. The multiple view approach has also been extended to the distributed paradigm, where it benefits from localized collaborative analysis [50, 51]. Again, however, these approaches are rarely, if ever, combined. The stereo approach suffers from the same occlusion effects as the monocular approach, and the multiple view approach relies either on 2D methods such as blob analysis which do not generalize well or on difficult and non-robust wide-baseline 3D reconstruction.

An approach employing distributed smart stereo cameras would impart a distributed multi-view method with robust true 3D scene reconstruction information, also sidestepping the occlusion problem, and as a result methods based on such a system would generalize very well.

### Scene Reconstruction

With the exception of some specific cases, two or more views are necessary for passive three-dimensional scene reconstruction. Generally, the entire purpose of stereo vision at its basic level is 3D scene reconstruction, and methods for this are well-established in the literature [33, 58, 59, 60]. Multiple views not strictly consisting of stereo camera rigs but employing the same or similar methods exist, usually generalized to a wide-baseline approach, but these methods are inherently not robust due to the difficulty of matching features across widely separated views [18, 52]. For many practical applications, more specific approaches allow for partial scene reconstructions which are sufficient for the application but do not generalize well. A review of

multi-ocular reconstructions with a focus on accuracy is presented in [53]. View synthesis is one possibility which happens to lend itself particularly well to the distributed paradigm, as explored in [54]; however, the drawback is that it is computationally expensive.

Distributed smart stereo cameras are an ideal solution for full 3D reconstruction of arbitrary scenes. Assuming a calibration method such as the one presented in this work is able to converge, local 3D reconstructions may be combined in a distributed fashion via collaboration between the appropriate nodes, allowing for fast, scalable, and generally applicable scene reconstruction methods.

### 2.1.2 Calibration

As was mentioned in Chapter 1, the majority of research in distributed smart cameras to date has focused on monocular vision at each node. A number of methods for distributed self-calibration have been proposed for this paradigm, and though the vision components are not readily applicable to 3D sensing nodes, the general localization and distribution concepts developed apply to any vision-based system.

The problem has been approached from one of two angles. Coming from the perspective of traditional sensor networks, the primary challenges are the directionality of vision sensors, the higher degree of accuracy required by vision applications, and the large volume of raw sensor data. Conversely, from the perspective of traditional computer vision, the challenge is in the scalable distribution of processing among nodes and the related limitations of network bandwidth.

While traditional sensor network methods generally employ omnidirectional sensors and thus require only localization, vision-based networks also require orientation. Traditional sensor networks have benefitted from the fact that their network topology is generally geographic (for example, with ad-hoc wireless communications), and thus their communication graph is a rough estimate of node localization. To apply similar methods to directional vision sensors, the concept of the *vision graph* is introduced in [6], where an edge on the graph represents shared field of view rather than a communication link.[1]

Functional calibration methods are presented for monocular distributed smart cameras in [6, 7]. These are based on wide-baseline stereo methods, which are generally not robust due to the matching problem [18], and require unwieldy initialization schemes or dictate deployment constraints. Some methods, such as [8], use motion of objects in the scene to calibrate, but these still suffer from the matching problem to a degree and require certain kinds of scene. Potentially more robust methods are presented in [9, 10]; however, these require the use of markers or beacons placed in the environment, which is infeasible in many cases and may constrain deployment or extension to dynamic calibration.

With the true 3D sensing network paradigm introduced in [1], advocating distributed smart stereo cameras, a calibration method called Lighthouse is presented in [2] which uses 3D features and geographic hash tables [4] to localize and orient nodes. The same basic concept is employed by the calibration method described in this work, but the method in [2] does not appear to converge well, provides little information on the vision and registration problems, and has some inherent inefficiencies which are addressed here.

[1] These concepts are used throughout this work, and are elaborated upon in Section 3.3.

## 2.2 Interest Point Detection

The detection of salient features in the three-dimensional environment is fundamental to this work. These features are triangulated from *interest points* or *corners* detected automatically in the two-dimensional images. The development of algorithms which do so repeatably, in such a way that the detected features are stable from various viewpoints, is an area of research which has received much attention lately. The literature on interest point detection is reviewed here with an emphasis on repeatability, which as will be seen in Section 5.2 is the primary concern in this work.

One of the most popular corner detection methods is that developed by Harris and Stephens [16], which operates by considering the differential with respect to direction of the intensity values of a small window surrounding each pixel, and choosing those which return high values in both directions as corners. The advantage of this method over previous attempts is that if a circular or circularly-weighted window is used, the response is isotropic. Thus, this detector is invariant to translations and rotations in the image plane. The Harris operator has been employed by a number of subsequent interest point detectors, such as SUSAN [17] and FAST [22, 23].

The Harris operator can also be made invariant to scale changes by computing a multi-scale measure at various scales [18]. It can also be made invariant to affine transformations in the image by iteratively warping the shape of the smoothing kernel or the image patch [18, 19].

Algorithms such as SIFT [20] and SURF [24] employ operations such as the Laplacian of Gaussian (LoG), Difference of Gaussians (DoG), and Determinant of Hessian (DoH) to detect *blobs*, which include interest points and edges. These operations are similarly made invariant to scale and affine transformations. This type of detector yields a large number of repeatable points with distinctive associated descriptors.

An evaluation of these detectors and their performance in the context of 3D objects can be found in [25].

## 2.3 Registration

*Registration* is the process of transforming different sets of data, normally acquired visually, into a common coordinate system. It is an active research topic in computer vision with a variety of applications, and many different methods have been developed suiting diverse needs. The point registration concept lends itself particularly well to the problem of determining relative motion from point features detected in the environment, so it plays a central role in this work.

A recent survey of registration algorithms [12] separates them into two classes: those which are able to operate with no initial alignment estimate but yield relatively inaccurate results, called *coarse* registration methods, and those which can only operate given an initial alignment estimate but accordingly generate highly accurate results, called *fine* registration methods. As will be seen in Section 4.3.2, the use of two registration stages (one coarse, one fine) is a pivotal part of this work. The specific algorithms in Section 5.3 were chosen based on the algorithm requirements from those reviewed in [12].

# Chapter 3

# Theoretical Foundations

## 3.1 Geometry

### 3.1.1 Euclidean Distance

The Euclidean distance is the intuitive straight-line distance between two points in a metric space. The distance between two points $P = (p_1, p_2, \cdots, p_n)$ and $Q = (q_1, q_2, \cdots, q_n)$ in $n$-dimensional Euclidean space is defined as:

$$||P - Q|| = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2} \tag{3.1}$$

Most of the distances encountered in this work are computed in 3D Euclidean space.

### 3.1.2 Rotation Matrix

A rotation matrix $\mathbf{R}$ in Euclidean $n$-space is a $n \times n$ real orthogonal matrix, whose transpose is its inverse, i.e. $\mathbf{R}^T = \mathbf{R}^{-1}$, and whose determinant $\det(\mathbf{R}) = 1$. A $3 \times 3$ rotation matrix corresponds to a geometric rotation about a fixed origin in three-dimensional Euclidean space. The product of two rotation matrices is itself a rotation matrix, corresponding to a composition of the rotations.

Rotating about the origin intuitively means (by convention) rotating by $\theta$ about the original $x$-axis from $y$ to $z$, rotating by $\phi$ about the original $y$-axis from $z$ to $x$, and rotating by $\psi$ about the original $z$ axis from $x$ to $y$, illustrated in Figure 3.1.

The rotation matrix for this set of angles can be found by composition (multiplication) of the individual matrices for each of these rotations, resulting in the following representation:

$$\mathbf{R} = \begin{bmatrix} cos\phi cos\psi & sin\theta sin\phi cos\psi - cos\theta sin\psi & cos\theta sin\phi cos\psi + sin\theta sin\psi \\ cos\phi sin\psi & sin\theta sin\phi sin\psi + cos\theta cos\psi & cos\theta sin\phi sin\psi - sin\theta cos\psi \\ -sin\phi & sin\theta cos\phi & cos\theta cos\phi \end{bmatrix} \tag{3.2}$$

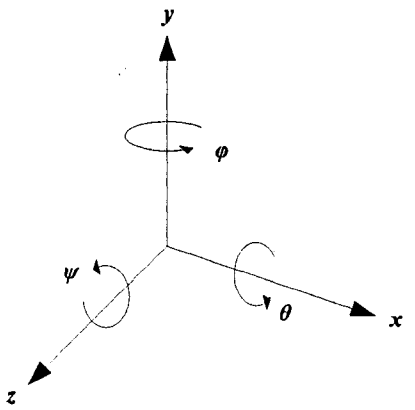


Figure 3.1: Fixed-Axis Rotation Convention

The angles in the range $[0, 2\pi)$ can also be recovered from the entries of the rotation matrix:

$$\phi = \begin{cases} \arcsin(-\mathbf{R}_{31}) & \text{if } \mathbf{R}_{31} \leq 0; \\ \arcsin(-\mathbf{R}_{31}) + \pi & \text{if } \mathbf{R}_{31} > 0. \end{cases} \tag{3.3}$$

$$\theta = \begin{cases} \arctan\left(\frac{\mathbf{R}_{32}}{\mathbf{R}_{33}}\right) + \pi & \text{if } \cos\theta = \frac{\mathbf{R}_{33}}{\cos\phi} < 0; \\ \arctan\left(\frac{\mathbf{R}_{32}}{\mathbf{R}_{33}}\right) + 2\pi & \text{if } \sin\theta = \frac{\mathbf{R}_{32}}{\cos\phi} < 0; \\ \arctan\left(\frac{\mathbf{R}_{32}}{\mathbf{R}_{33}}\right) & \text{otherwise.} \end{cases} \tag{3.4}$$

$$\psi = \begin{cases} \arctan\left(\frac{\mathbf{R}_{21}}{\mathbf{R}_{11}}\right) + \pi & \text{if } \cos\psi = \frac{\mathbf{R}_{11}}{\cos\phi} < 0; \\ \arctan\left(\frac{\mathbf{R}_{21}}{\mathbf{R}_{11}}\right) + 2\pi & \text{if } \sin\psi = \frac{\mathbf{R}_{21}}{\cos\phi} < 0; \\ \arctan\left(\frac{\mathbf{R}_{21}}{\mathbf{R}_{11}}\right) & \text{otherwise.} \end{cases} \tag{3.5}$$

### 3.1.3 Relative Pose

*Pose* is a concept used to describe the relative motion between two nodes of a distributed smart camera network, which is the basis of calibration. Each node is considered to have its own local coordinate system. The *relative pose* of node $A$ with respect to node $B$ is denoted $P_{AB}$, and is the rigid transformation in 3D Euclidean space from the coordinate system of $A$ to that of $B$.

The transformation $P_{AB} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ consists of a rotation matrix ($3 \times 3$ real orthogonal matrix) $\mathbf{R}_{AB}$ and a 3-element translation vector $\mathbf{T}_{AB}$. $P_{AB}$ maps a point $x \in \mathbb{R}^3$ as follows:

$$P_{AB}(x) = \mathbf{R}_{AB}x + \mathbf{T}_{AB} \tag{3.6}$$

### Identity

The *identity pose* is represented by $P_I = (\mathbf{R}_I, \mathbf{T}_I)$. The transformation associated with this pose has no effect, so that $P_I(x) = x$. $\mathbf{R}_I$ and $\mathbf{T}_I$ are defined as follows:

$$\mathbf{T}_I^T = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \tag{3.7}$$

$$\mathbf{R}_I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.8}$$

### Inversion

The inverse of pose $P = (\mathbf{T}, \mathbf{R})$, denoted $P^{-1}$, which maps the destination coordinate basis back to the source, can be determined as follows:

$$\begin{aligned} P(x) &= \mathbf{R}x + \mathbf{T} \\ -\mathbf{R}x &= -P(x) + \mathbf{T} \\ \mathbf{R}x &= P(x) - \mathbf{T} \\ x &= \mathbf{R}^{-1}P(x) - \mathbf{R}^{-1}\mathbf{T} \\ x &= \mathbf{R}^{-1}P(x) - \mathbf{R}^{-1}\mathbf{T} \\ P^{-1}(P(x)) &= \mathbf{R}^{-1}P(x) - \mathbf{R}^{-1}\mathbf{T} \\ P^{-1}(x) &= \mathbf{R}^{-1}x - \mathbf{R}^{-1}\mathbf{T} \end{aligned} \tag{3.9}$$

### Composition

A succession of pose transformations $P_{BC}(P_{AB}(x))$ can be composed into a single pose, denoted $(P_{AB} \circ P_{BC})(x)$, as follows:

$$\begin{aligned} P_B(P_A(x)) &= \mathbf{R}_B(\mathbf{R}_A x + \mathbf{T}_A) + \mathbf{T}_B \\ (P_A \circ P_B)(x) &= \mathbf{R}_B\mathbf{R}_A x + (\mathbf{R}_B\mathbf{T}_A + \mathbf{T}_B) \end{aligned} \tag{3.10}$$

This transformation maps from the coordinate system of $A$ to that of $B$, then from that of $B$ to that of $C$; therefore, the transformation from $A$ to $C$ can be computed via composition as $P_{AC} = (P_{AB} \circ P_{BC})(x)$. This operation is transitive, so one node's pose relative to another can be computed indirectly over an arbitrary number of intermediate poses if they exist.

## 3.2 Node Concepts

### 3.2.1 Nodes and Groups

A *node* is the abstract or physical smart stereo camera device itself; nodes shall be denoted by sequential capital letters ($A$, $B$, and so forth). The set of all nodes in the network is denoted $\mathbb{N}$ (where $|\mathbb{N}|$ represents the total number of nodes).

A *group* is a set of nodes which agree on a single *leader* node; a group led by node $A$ shall be denoted $G_A$ (where $|G_A|$ represents the number of nodes in the group). Every group is a subset of the network ($G_A \subseteq \mathbb{N}$), and every node is a member of exactly one group (so, if $G_A$ and $G_B$ are two separate groups, $|G_A \cap G_B| = 0$).

### 3.2.2 Node Pose Conventions

The relative pose, as defined in Section 3.1.3, is primarily used to describe either the unknown true pose of one node relative to another or the final pairwise pose estimate as estimated by the calibration process. As stated, the pose of node $A$ relative to node $B$ is denoted $P_{AB}$.

In this work, an intermediate type of relative pose is employed to describe the results of coarse registration and grouping. The *relative coarse pose estimate* of node $A$ with respect to node $B$ is denoted $C_{AB}$. Additionally, within a group, it will be seen that each node has a *group coarse pose estimate* relative to the leader of the node's current group; thus, a node $A$ that is a member of group $G_B$ has a group coarse pose $C_A$, which is equal to $C_{AB}$.

### 3.2.3 Point Sets and Features

A *point set* is the full set of interest points detected locally at a node; the point set of node $A$ shall be denoted $S_A$. The *overlap* between point sets $S_A$ and $S_B$ refers to the size of the intersection of the two sets $|S_A \cap S_B|$, said intersection occurring where a point in $S_A$ corresponds to the same physical point as a point in $S_B$. The *percent overlap* is defined as follows:

$$\%O(S_A, S_B) = \frac{|S_A \cap S_B|}{\max(|S_A|, |S_B|)} \times 100\% \tag{3.11}$$

A *feature* is any subset of the point set of a certain size (determined by a parameter of the algorithm); when discussing a single arbitrary feature from node $A$, it shall be denoted $F_A$, where $F_A \subseteq S_A$. Two features $F_A$ and $F_B$, from nodes $A$ and $B$ respectively, are considered to *match* (denoted $F_A \approx F_B$) if each point in $F_A$ corresponds to the same physical point as a point in $F_B$. In the context of the algorithm, it is impossible to ascertain this correspondence, so the term *match* implies rather a presumed match based on a criterion of geometrical similarity.

## 3.3 Graphs

Three types of undirected graphs are helpful in describing distributed smart camera calibration: the *communication graph*, the *vision graph*, and the *calibration graph* [6]. Graphs are described as *connected* if there exists a path connecting every pair of nodes, and *complete* if there exists an edge between each pair of nodes.

### 3.3.1 Communication Graph

The communication graph describes the effective communication links between nodes in the network from the perspective of the layer presented to the application. A complete communication graph indicates that any node may communicate directly with any other node, whether physically or via lower-level routing software. This graph is of limited interest here due to the abstract network assumption (Section 4.2.5).

### 3.3.2 Vision Graph

The vision graph describes which nodes share significant portions of their field of view. A pair of nodes have a connecting edge in this graph if the volume of space in the intersection of their fields of view is considered large enough that it might contain sufficient data for the operations required by the algorithm; this is, of course, dependent on a large number of factors and in general the vision graph is used in a qualitative context.

Figure 3.3 demonstrates a vision graph associated with the nodes with overlapping fields of view shown in Figure 3.2.

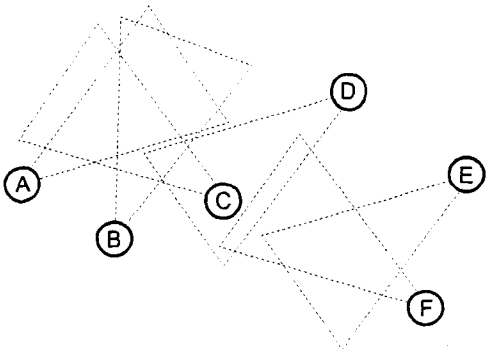


Figure 3.2: Example Field of View Overlap

### 3.3.3 Calibration Graph

The calibration graph describes which nodes have a direct estimate of their pairwise pose. Obviously, it is desirable that this graph be connected, so that any two nodes $X$ and $Y$ can compute their relative pose $P_{XY}$ by

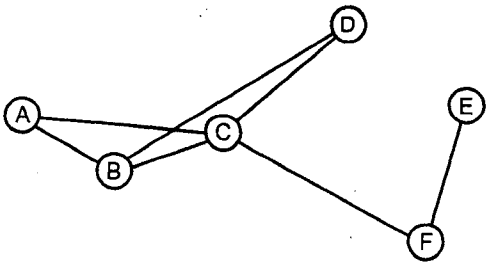


Figure 3.3: Example Vision Graph

composition of known pose estimates. Edges can only be established where there exist edges in the vision graph, and the most complete calibration graph possible is identical to the vision graph. For later reference, in this work, edges are established by pairwise pose refinement, the second stage of calibration (see Section 4.3.2).

Figure 3.4 demonstrates a calibration graph that might arise from nodes with the vision graph of Figure 3.3. In this case, nodes *B* and *D* do not have an edge between them, and so do not have direct pairwise pose estimates to one another. They would need to compute an indirect estimate, possibly through node *C*, to perform any tasks requiring calibration.

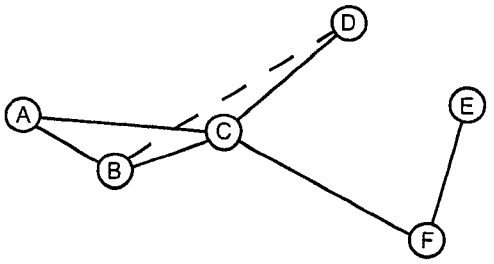


Figure 3.4: Example Calibration Graph

# Chapter 4

# General Solution

## 4.1 Problem Statement

The overall objective is to spatially calibrate a series of homogeneous smart stereo camera nodes, with no *a priori* knowledge and using only the nodes' 3D visual data, in a distributed fashion. Assuming the visual data consists of a set of 3D points triangulated from stereo images of the environment, the problem may be reduced to geometrical terms:

> Given a set of nodes $\mathbb{N}$, each node $X \in \mathbb{N}$ having a point set $S_X$, estimate the pose $P_{XY}$ for enough node pairs $(X,Y)$ such that the calibration graph for $\mathbb{N}$ is connected.

The shared view assumption (Section 4.2.2) and the repeatability criterion of interest point detection (Section 5.2.1) imply a sufficient degree of overlap between a sufficient number of node pairs for convergence.

This chapter will develop a practical, network-based, distributed algorithm for solving this problem without considering the details of certain parts, such as the actual acquisition of the 3D point sets using cameras or the noise-tolerant registration of one node's point set to another's. It will be described in a modular fashion, so that different specific methods may be chosen, where applicable. The design is, of course, coupled to a degree with extant algorithms, but as much as possible these considerations are relegated to Chapter 5, which discusses the practical implementation.

## 4.2 Assumptions

### 4.2.1 Pre-Deployment Offline Access

It is assumed that, prior to deployment of the network, there is a period during which each node may be accessed without restriction in a controlled environment, in order to perform certain essential modifications to software (such as assignment of a unique identifier, network configuration, and intrinsic/stereo calibration

of the cameras). This could occur as part of the manufacturing process, so it does not preclude the node devices being ready for deployment "out of the box" as is desired.

### 4.2.2 Shared View

For full convergence, it is assumed that the vision graph is connected. This imposes the basic minimum constraint on node deployment that the shared field of view of the entire network must be continuous and have substantial internal pairwise overlap.

### 4.2.3 Fixed Nodes

It is assumed that each node is fixed in its location and orientation relative to all other nodes. It is also assumed that, once internally calibrated for stereo vision, no node changes the relative motion between its cameras or the internal parameters (e.g. focal length) of either of its cameras. In theory, such changes can be locally detected and measured, and then compensated for in some way; however, that is beyond the scope of this work.

### 4.2.4 Static Scene

It is assumed that the contents of the scene are fully static for the purposes of acquiring calibration point sets. This can be restated as an assumption that the scene contents giving rise to each node's calibration image pair are identical. This assumption is made solely for simplicity, and it could easily be relaxed using background estimation techniques (widely researched and generally used for foreground object segmentation) or accurate temporal synchronization.

### 4.2.5 Abstract Network

It is assumed that the nodes are capable of autonomously forming an ad-hoc network wherein each node can be addressed by a unique identifier. This carries with it an implicit deployment assumption about the network medium; for example, depending on the specific requirements of the medium, the nodes are each wired to a network trunk or hub, or are all within wireless range of at least one neighbour. No specific media or protocols are prescribed or assumed within this work, so any configuration which does not violate the aforementioned assumption and has sufficient capacity is acceptable.

From the algorithm's point of view, the network is assumed to be *fully connected* [57], or in other words, the communication graph is assumed to be complete. As mentioned in Section 3.3, this does not imply anything additional about the physical topology of the network; it simply means that all necessary hardware and software layers exist so that the following criteria are met:

1. Each node has an identifier that is unique within the network.

2. Each node maintains a list of all other nodes' identifiers.

3. Each node may address any other node by its identifier, and send it an arbitrary amount of data with assured delivery.

These assumptions are valid for most modern networks. The first two requirements are provided at the data link and network layers, and the third at the transport layer. The algorithm itself is therefore independent of the implementation detail of interfacing to the underlying network.

### 4.2.6 Local Assumptions

These assumptions apply only in the development of the algorithm within this chapter; suitable implementations are discussed in Chapter 5.

#### Stereo 3D Vision

It is assumed that a suitable set of 3D points are detected at each node. The algorithm is described without regard for the details of acquisition, so in essence, it is assumed that each node has calibrated its cameras for stereo vision reasonably and can perform repeatable interest point detection, correspondence, and triangulation. Regardless of how it is accomplished, the calibration method expects a set of 3D points relative to the node's local coordinate system to be available throughout its execution.

To implement this assumption in a practical case, 3D point sets can be manually supplied to the algorithm, allowing for control over the degrees of repeatability and error. This local assumption is removed by examining and introducing specific algorithms for calibration, interest point detection, correspondence, and triangulation in Sections 5.1 and 5.2.

#### Abstract Registration

It is assumed that the algorithm has access to suitable coarse and fine registration algorithms, as explored in Section 4.3.2. The development of the calibration method is, of course, informed by knowledge of existing registration algorithms, but prescription will be limited to general classes of algorithms for modularity. The calibration method expects the coarse registration algorithm to take two point sets and some parameters as input and return a relative pose and an error metric, and the fine registration algorithm to take two point sets, an initial rotation estimate, and some parameters as input and return an accurate relative pose and an error metric.

This local assumption is removed by examining and introducing specific algorithms for coarse and fine registration in Section 5.3.

## 4.3 Problem Analysis

In this section, the fundamental operation of the calibration method is developed progressively based on the requirements and assumptions, finally resulting in an outline for a general solution. The progression reflects the thought process and research behind the theoretical development of the method.

### 4.3.1 3D Visual Data Primitive

The most fundamental piece of three-dimensional visual data that can be obtained by a stereo camera node is a point. A 3D point in the coordinate system of the observing node is a single piece of range image data, triangulated from corresponding 2D points in the stereo image pair using known calibration parameters. Thus, the point is the basic data primitive used by the calibration method. One of the local assumptions in Section 4.2.6 ensures that a number of such points are detected at each node; the point set represents the entirety of any node's visual observations for the purposes of developing the calibration method.

### 4.3.2 Two-Stage Registration

As discussed in Section 2.3, registration algorithms may be divided into two types: coarse registration methods, which do not require an initial alignment estimate and produce relatively inaccurate results, and fine registration methods, which require an initial alignment estimate and produce accurate results.

In this case, no alignment estimate is initially available for the nodes' point sets, yet accurate pose estimates are desired. The typical solution is a two-stage registration, where a coarse algorithm is used first to get an alignment estimate, and then this estimate is supplied to a fine algorithm. Fundamentally, this approach is applicable, but on closer inspection there is still a problem. Not only is there no intial alignment estimate, but the nodes do not even know whether or how their point sets overlap with those of the other nodes. All registration algorithms depend on some substantial degree of overlap between the two data sets, so some method of determining what data sets to operate on is necessary; this is the *feature matching* process described in the following section.

### 4.3.3 Feature Matching

In order to find coarse pose estimates between nodes with no knowledge of their point set overlap in a distributed fashion, a pairwise feature matching process similar to that described in [2] can be employed. Following from Section 4.3.2, a *feature*, in this context, is a fixed-size subset of a node's 3D point set.

The goal is to find pairwise matches between nodes' features, and then use those matches to calculate coarse relative pose estimates for the node pairs. The former part can be achieved by applying a coarse registration algorithm on similar features and observing if the resulting error metric falls below a certain threshold. Conveniently, the same algorithm also outputs a relative pose estimate, so the basis for the latter part of the goal occurs simultaneously.

### Feature Selection

Periodically,[1] a feature, or subset of fixed size $f$, is selected[2] from the point set (illustrated in Figure 4.1), and compared to similar features from other nodes. For now, the details of how they are brought together for comparison are ignored.

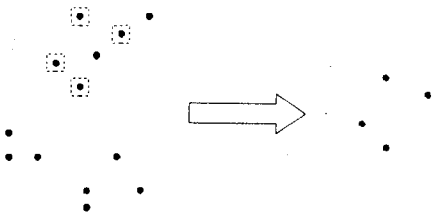

Figure 4.1: Feature Selection ($f = 4$)

Consider point sets from two nodes, $A$ and $B$, from which, according to the coarse matching algorithm, each node randomly selects a feature (non-repeating combination), resulting in $F_A \subseteq S_A$ and $F_B \subseteq S_B$ where $|F_A| = |F_B| = f \leq |S_A \cap S_B|$. The performance of the matching scheme depends directly on the probability of a match between $F_A$ and $F_B$, $P(F_A \approx F_B)$, since as this probability decreases, it takes longer to find matches and more bandwidth and storage must be used to transmit and store categorized features.

First, the individual probabilities that $F_A$ and $F_B$ will be within the set of shared points $S_A \cap S_B$ are found. Let $P(Q) = P(F_A \subseteq S_A \cap S_B)$ and $P(R) = P(F_B \subseteq S_A \cap S_B)$, and let $C(n,k)$ represent the binomial coefficient, indicating the number of possible non-repeating combinations of size $n$ chosen from $k$.

$$P(Q) = \frac{C(|S_A \cap S_B|, f)}{C(|S_A|, f)} \tag{4.1}$$

$$P(R) = \frac{C(|S_A \cap S_B|, f)}{C(|S_B|, f)} \tag{4.2}$$

$$\begin{aligned} P(Q \cap R) &= P(Q)P(R) \\ &= \frac{C(|S_A \cap S_B|, f)^2}{C(|S_A|, f)C(|S_B|, f)} \end{aligned} \tag{4.3}$$

[1] The delay between feature selections is determined by how quickly nodes are generally able to describe, propagate, and match features. The delay should be as short as possible while avoiding network flooding and a backlog of features to be matched at the nodes. Due to the complexity of modeling the various parameters that might have an effect on this, in the current implementation, it is optimized by trial and error.

[2] Different selection methods are possible. One might select features randomly, as in [2], or precompute them and select them in some meaningful order, as described in Section 4.4.2. Also, even with no *a priori* knowledge of the scene structure, the feature matching scheme might benefit in performance from the application of some constraints on feature selection. A trade-off between matching probability and overall convergence might be optimized in this way. Constraints on the geometric nature of possible features might also allow for better performance in feature categorization (discussed later in this section). Feature selection constraints have not been investigated in depth in this work.

Assuming that conditions $Q$ and $R$ are satisfied (that is, all points in features $F_A$ and $F_B$ are shared in $S_A \cap S_B$), for a given $F_A$, only one combination $F_B$ will match it.

$$P(F_A \approx F_B | Q \cap R) = \frac{1}{C(|S_A \cap S_B|, f)} \tag{4.4}$$

From Equations 4.3 and 4.4, $P(F_A \approx F_B)$ can be calculated.

$$\begin{aligned} P(F_A \approx F_B) &= P(F_A \approx F_B | Q \cap R) P(Q \cap R) \\ &= \left[\frac{1}{C(|S_A \cap S_B|, f)}\right]\left[\frac{C(|S_A \cap S_B|, f)^2}{C(|S_A|, f) C(|S_B|, f)}\right] \\ &= \frac{C(|S_A \cap S_B|, f)}{C(|S_A|, f) C(|S_B|, f)} \\ &= \frac{|S_A \cap S_B|! f! (|S_A| - f)! (|S_B| - f)!}{|S_A|! |S_B|! (|S_A \cap S_B| - f)!} \end{aligned} \tag{4.5}$$

By inspection, $P(F_A \approx F_B)$ increases as the ratio of $|S_A \cap S_B|$ to $|S_A|$ and $|S_B|$ increases, for a given $f$. It will be seen in Section 5.2.1 that this ratio is the *repeatability* criterion of interest point detection. The number of correspondences $|S_A \cap S_B|$ and the repeatability score are controlled by the interest point detection algorithm and its parameters.

The *feature size* parameter $f$, however, must be chosen directly, and there is no quantitative optimum that works in all cases. Decreasing $f$ increases the probability of false matches, which impacts convergence; obviously, however, for unique 3D matching, there is a condition that $f \geq 3$. It may also result in less accurate estimates for each match in coarse registration. On the other hand, increasing $f$ has a negative impact on matching performance and possibly on the later fine registration, as it drives up the minimum required number of correspondences $|S_A \cap S_B|$ which means, as will be seen in Section 5.2, that $|S_A|$ and $|S_B|$ must be greatly increased, reducing matching performance directly as well as possibly reducing repeatability. Increasing $f$ may also impact convergence if the resultant $|S_A \cap S_B|$ requirement begins to exceed the actual point set overlap of nodes that might otherwise have matches.

### Feature Categorization

In the previous subsection, no details are given as to how features are brought together for comparison. This is an important issue in terms of parallelization and scalability, as there is no central place where all features can be compared.

The idea of feature categorization is borrowed from the data-centric storage literature, used with reference to distributed smart camera networks in [1] and more specifically to their calibration in [2]. The goal is to evenly distribute the processing and storage of the data in a distributed system based on some quantitative or qualitative metric of the data itself. Since, for obvious reasons, it is desirable in this case to compare features which are geometrically similar, the logical choice is some descriptor of the geometry of a feature invariant to translation and rotation, called the *geometric descriptor* function (equivalent to the *geometric hash* in [2]).

This metric is then hashed through a categorization function, which returns the address(es) of the destination node(s) for the feature. In [2], the categorization function is a *geographic hash* used to store the feature in a geographic hash table (GHT) [4], but as the authors state, this requires localization, which is part of what the calibration method is attempting to determine to begin with. Their solution is to bootstrap ever-larger GHTs by feature injection and merging; however, this results in a lower convergence rate, and introduces categorization problems by making the hash table dynamic. To improve the convergence rate of this type of scheme, one might consider starting with a network-wide GHT using a geographic routing method which does not require localization, such as [5]. However, there are several reasons why this is not done here:

1. Geographic routing methods require specific network topologies and protocols, which violates the abstract network generalization (Section 4.2.5).

2. There is no reason to believe that geographic hashing is the only data-centric storage method suitable for distributing features for comparison.

3. Geographic hashing may, in fact, be ill-suited to feature categorization, as consistent hashing is difficult, especially in a dynamic table.[3]

A better solution is to use a consistent and evenly-distributed data-centric storage technique which incorporates the entire network from the beginning, one which does not require localization or any other information the nodes do not initially possess. The abstract network assumption (Section 4.2.5) states that nodes are aware of and able to route messages to all other nodes in the network. Since the features are to be distributed based on their local geometric structure, in general, it is appropriate to divide the solution set of the geometric descriptor function equally among the nodes in the network (with some overlap, depending on the measurement accuracy, so that similar features at boundaries are compared). If this is done in the same way at each node, then geometrically similar features can be stored at the same node regardless of the source, satisfying the objective.

Figure 4.2 shows how features are categorized. In this example, nodes $A$ and $C$ both detect the feature represented by the square. They individually compute the geometric descriptor of this feature, and the results are very close (ideally identical) because, of course, the hash function is operating on the same feature and is invariant to translation and rotation. They both, therefore, send their individual features, as they view them, to the same node; in this case, node $B$. Node $C$ is shown detecting another feature as well, represented by the triangle. This feature is substantially different in geometry, so the geometric descriptor falls in the range of a different node, $E$, to which $C$ sends this feature.

The geometric descriptor function must be *deterministic*, so that differing results from the same function imply differing input features. It is not necessarily *injective*, so it may be possible for different input features to generate the same result. Normally, it is desirable for hash functions to have the *mixing property*, meaning that a small change in the input results in a large change in the result; this is unacceptable in this case. The

[3] As explained in [2], an appropriate range would need to be selected for the GHT so that features are spread out evenly, and this would likely require an irregular hashing function shape. Furthermore, with the tables constantly growing, features would frequently need to be recategorized and forwarded to new nodes.

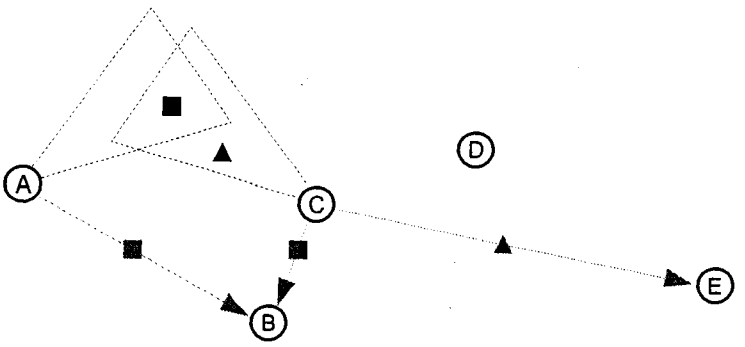


Figure 4.2: Feature Categorization

geometric descriptor is used to gauge similarity between features, so instead, it should be a *smooth* function, and ideally a linear one, in the feature geometry.

Ideally, the difference between the descriptors of two features $F_A$ and $F_B$ describes the degree of difference $d$ between those features (Equation 4.6):

$$d(F_A, F_B) = |g(F_A) - g(F_B)| \tag{4.6}$$

Based on the measurement accuracy of a node and the specific coarse registration algorithm used, there is a *similarity threshold* $t_d$, such that it is necessary to compare two features $F_A$ and $F_B$ if $d(F_A, F_B) < t_d$, and unnecessary otherwise; this will be termed the *similarity condition*. When the geometric descriptor space is divided among the nodes in the network, initially, there is no overlap at the boundaries. It is necessary to extend the range of the categorization boundaries of each node exactly $t_d/2$ in all directions, so that any two features less than $t_d$ different will be categorized together on at least one node (and will thus be tested for matching). Additionally, within a node, an incoming feature $F_A$ need only be tested for matching to a feature $F_B$ if the similarity condition is met.

The remaining problem is to obtain a more or less even distribution of the features (and thus, the processing load) when the geometric descriptor space is divided initially. This depends on the nature of the geometric descriptor. Since it is impossible for the nodes to know the actual solution set of the function, since that would require it to have copies of all possible features from all nodes and therefore defeat the purpose of categorization, an estimation of the feature density within the solution space of the descriptor function is required. This cannot depend on the locally detected features in any way; otherwise, each node would calculate the distribution differently. Without knowledge of the particular deployment environment, a generally acceptable distribution based on "average" feature geometry must be determined in advance, informed by the interest point detector used, the size of the node's field of view, and any constraints on feature detection.

### Match Reporting

When a feature is selected, categorized, and sent to the appropriate node for matching, it is represented in the local coordinate basis of the originating node, and tagged with that node's unique identifier so that the matching node knows which node detected it. The matching node then stores it in a local database, and attempts coarse registration against other features in its database which satisfy the similarity condition.

When a match between two features is found, the match is reported to both of the originating nodes. The report informs each node of the unique identifier of the node it matched features with, as well as the estimated relative pose of that node, based on the results of coarse registration. These are then used for grouping, which is covered in the following section.

To be more specific, coarse registration algorithms operate on a *model* point set and a *data* point set, and output a pose estimate $P$ of the data relative to the model. In this case the data is an incoming feature, and the features stored in the matching node's database serve as successive models.[4] When a match is found, the originating node of the model is sent the pose $P$, which, if the match is not a false one, is an estimate of the pose of the originating node of the data relative to the originating node of the model. The originating node of the data, however, needs the inverse of this pose, so instead of $P$, it is sent $P^{-1}$ as defined in Equation 3.9.

### Coarse Pose Estimation

While it is desirable to use a relatively small value for the feature size $f$, this results in a substantial possibility of false matches, especially in environments that contain similar objects which generate nearly identical features. Furthermore, even with a true match, the coarse registration algorithm may be unable to guarantee the degree of accuracy desired for a coarse pose estimate.

These problems can both be resolved by combining and averaging the results of several feature matches. To ensure that no false matches are used, a *match threshold* $t_m \geq 3$ is required as the minimum number of pairwise feature matches (as received by one node of the pair) whose pose estimates are sufficiently similar, with the entire set meeting the condition in Equation 4.7. To increase confidence in the result at a cost of reduced convergence, $t_m$ may be increased.

$$C_i \circ C_j^{-1} \approx P_l \tag{4.7}$$

This can be enforced by means of a *consistency threshold*, $t_c$, expressing the maximum Euclidean distance between points mapped by the match poses. Since the average of the suitable poses needs to be calculated for later use anyway, a simple implementation involves mapping a point – for generally good results, the centroid of the node's point set, $\mu_S$, can be used – through the average pose, then ensuring that each of the original poses maps the same point to within $t_c$ of that mapping in the Euclidean distance, as follows:

$$||C_m(\mu_S) - C_{avg}(\mu_S)|| \leq t_c \tag{4.8}$$

There is no readily apparent way to guarantee that all nodes which are potential candidates for pairwise pose

[4] This arrangement is purely by convention and could easily be reversed.

refinement (fine registration) have a coarse pose estimate without exhaustively matching all features, which is inefficient and probably infeasible. This critical issue is addressed by the *coarse grouping* scheme described in the following section.

### 4.3.4 Coarse Grouping

The goal of feature matching is to bring all of the nodes in the network into (coarse) alignment with one another. This does not necessarily require pairwise matches between all nodes: one node might obtain its coarse pose with respect to another node either directly, or indirectly as a composition of mappings (according to Equation 3.10) over a number of hops across nodes. This is the purpose of groups and the group coarse pose (defined in Sections 3.2.1 and 3.2.2, respectively).

The benefit of grouping is that once nodes are within the same group, there is no longer any need to perform feature matching and coarse pose estimation between those nodes. The entire network can therefore be brought into alignment with one another by a process of merging groups. If the merging process is designed such that the coarse pose estimates within a group are acceptably accurate for fine registration (discussed later in this section), then any two nodes which, based on these estimates, share a significant portion of field of view, regardless of whether they directly estimated their coarse pose relative to one another, can refine their pairwise pose. Groups are therefore sufficient to ensure that *all* candidate pairs undergo pose refinement, with no need for exhaustive feature matching.

Groups might conceivably be implemented in a number of different ways, but this calibration method requires it to be distributed and homogeneous.

#### Group Merging and Leaders

In [2], nodes are initialized into a *singleton* GHT: a degenerate case containing only the one node. It is sensible, in keeping with the homogeneity requirement and distributed paradigm, that all nodes begin in singleton groups as well. Note that a singleton group is in fact a true group (though also a degenerate case) according to the definition, since the node's pose relative to itself is simply the identity pose, as represented by Equations 3.7 and 3.8.

When two singleton groups find an acceptable coarse pose estimate relative to one another based on feature matching, they can merge into a single group containing both nodes, as they then collectively satisfy the definition of a group. However, the details of how this is to be achieved are not readily apparent. Obviously, the nodes need to agree on a method for exchanging the coarse pose information, but they must do so in a distributed fashion. Further problems arise when one considers what happens when a third node enters the group, based on a relative coarse pose estimate with one of the original nodes. A naïve approach might involve each node storing pairwise estimates with respect to all other nodes in the group, but this is inefficient in terms of scalability.

To resolve this, the concept of a *group leader* is introduced. The leader is simply a node within any given group whose local coordinate basis serves as the basis for every other node's group coarse pose estimate. The leader's own pose estimate, of course, is the identity pose. Group leaders also provide an elegant method for

group nomenclature: the identifier of the leader doubles as the identifier of the entire group. This intrinsically indicates which node in a group is its leader.

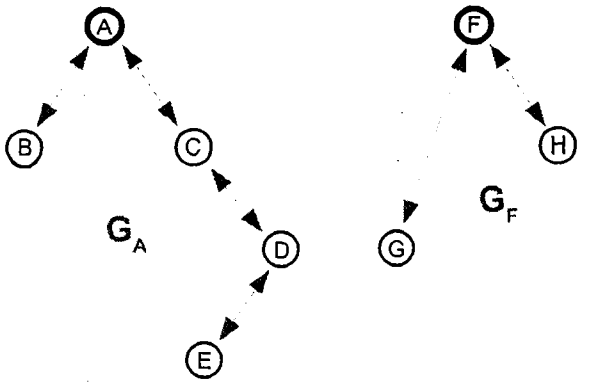


Figure 4.3: Group Topology

Figure 4.3 shows the conceptual topology of two groups, $G_A$ and $G_F$. The double-ended arrows represent actual pairwise coarse pose estimates obtained via feature matching, and thus the path, or hops, by which each node's coarse pose estimate relative to the group leader was originally computed. In reality, only the group coarse pose estimate relative to the current leader need be stored (implementation concerns notwithstanding). Note that although a node such as $E$ might well share features with node $C$ or even node $A$ and thus could obtain a more direct (and less erronous) group coarse pose estimate, once it is within group $A$, it will not do so. As mentioned previously, this arrangement would seem to give rise to potential accuracy issues; these are addressed later in this section.

Initially, the nodes in singleton groups are, by definition, the leaders of those groups. When two groups merge, a new leader must be chosen from one of the two original group leaders,[5] and the new leader's original group essentially annexes the other group.

The merging process itself is simply a matter of bringing the pose estimates of the merging group's nodes into the coordinate basis of the annexing group. Since any pair of nodes (one in each group) can initiate a merge based on their discovery of a pairwise coarse pose estimate by feature matching, those nodes must be capable of figuring out on their own how to carry out the merge. Each node is thus endowed with the following information related to its current group:

- The identifier of its current group (and thus, its leader).
- Its coarse pose estimate relative to its group.

[5]Technically, any node in either of the original groups could become the new group's leader, by the same mechanism, since the nodes could all determine their coarse pose relative to any such node. However, choosing one of the nodes already occupying the position is vastly less complex in practice.

- A list of other nodes in its group.

The merging process consists entirely of updating these three pieces of information, which can be done (for the entire group) by the initiating nodes themselves. The annexing group simply updates its list of nodes to include the nodes in the new group, while the merging group also updates to the new group identifier and changes coarse pose estimates accordingly.

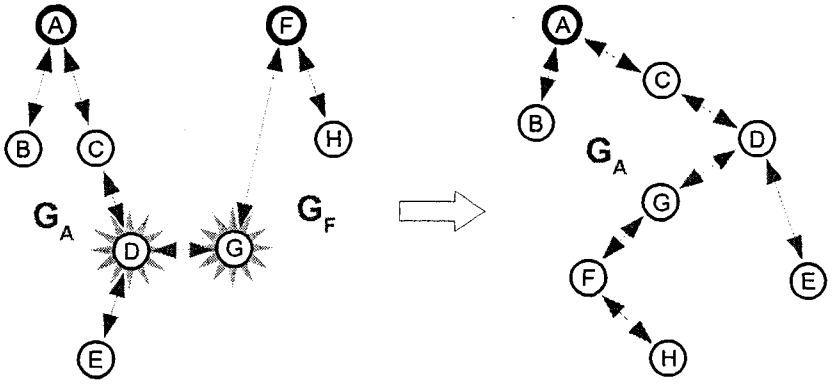


Figure 4.4: Group Merging

Figure 4.4 illustrates a typical group merge. Node $D$, of group $G_A$, and node $G$, of group $G_F$, find a relative coarse pose estimate through feature matching, and initiate a merge. The nodes in group $G_A$ do not modify their group coarse pose information. Node $G$'s new group coarse pose estimate ($C'_G$) is the composition of its estimated pose relative to node $D$ with node $D$'s group coarse pose estimate:

$$C'_G = C_{GD} \circ C_D \tag{4.9}$$

The new group coarse pose estimates for the merging group's leader (node $F$) and any other nodes in the merging group (in this case, node $H$) can similarly be calculated as compositions of known pose estimates:

$$C'_F = C_G^{-1} \circ (C_{GD} \circ C_D) \tag{4.10}$$

$$C'_H = C_H \circ (C_G^{-1} \circ (C_{GD} \circ C_D)) \tag{4.11}$$

A node will choose to employ Equation 4.9, 4.10, or 4.11 depending on whether it is the initiating node, the group leader, or another node, respectively.

Since merging consists of composition operations, it is a transitive operation which can occur based on matches (and the resultant relative coarse pose estimates) between any pair of nodes in different groups.

#### Alignment Accuracy

Since coarse pose estimates are used exclusively as a precursor to the fine registration stage, these estimates need only be accurate enough to provide an acceptable basis for the fine registration algorithm employed. These are generally quite forgiving, able to perform well with initial relative rotations of up to 30°, and direct coarse registration results with good data are generally well within this range [12]. The coarse registration results are also likely to be especially accurate in this case, since the feature matching scheme ensures that there are no outliers.

However, the error accumulated over multiple hops may easily push the initial estimate outside the fine registration algorithm's acceptable range. It is improbable that two nodes observing enough of the same points to initiate pairwise fine registration will have enough hops between them to accumulate this degree of error, but that is by no means a guarantee. No provisions are made for this possible problem here, but it may need to be considered in some situations, for example by enforcing a maximum path length for group coarse pose estimates.

### 4.3.5 Pairwise Pose Refinement

Once a given pair of nodes belong to a group via the feature matching process, those nodes can use their coarse relative pose estimate as a starting point for pose refinement. This is achieved by applying a fine registration algorithm to a large number of points initialized into coarse alignment.

#### Limiting Fine Registration Points

In general, supplying a larger number of corresponding points supplied to the fine registration algorithm will yield a more accurate fine pose estimate. A naïve approach would simply have one node send its entire point set to the other for fine registration; this would certainly maximize the total number of points (and therefore, the total number of corresponding points), but extraneous points due to differing field of view would reduce the correspondence ratio, or overlap, between the sets, negatively affecting the performance of registration. Also, since such an approach is not initially informed by the coarse pose estimates, there is no inherent way to decide whether nodes share any field of view at all, and thus many computationally intensive fine registrations would be performed for no purpose.

It is therefore desirable to limit in advance the exchanged points to those which could possibly be found within the shared field of view of both nodes. A simple and effective way to achieve this is to estimate the field of view of each node as a cone extending along its local $z$-axis to a certain distance, determining the intersection space of the cones as transformed by the coarse relative pose estimate, and selecting only those points within the intersection space. This is depicted in two dimensions in Figure 4.5, where nodes $A$ and $B$ would attempt pose refinement using only the indicated points, and node $C$ would not attempt it with either node $A$ or node $B$ at all.

Since only a coarse pose estimate is available, and also due to the effects of occlusion and instability in interest point detection, any such solution is inherently rough, but at the very least a potentially large number

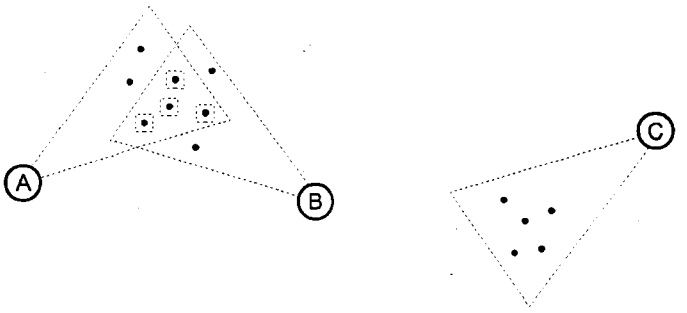


Figure 4.5: The Field of View Cone

of pointless fine registrations can be avoided.

### Pose Accuracy

Pose estimates returned by the fine registration algorithm should be rejected if the registration error exceeds a certain threshold.

### Indirect Pairwise Pose

The goal of the calibration algorithm is to provide a means for *any* pair of nodes in the network to determine, with minimal computation, an accurate relative pose to one another. Clearly, in the general case, not all nodes will share enough field of view or corresponding points to perform direct pairwise pose refinement. One immediately obvious solution is to repeat the composition procedure used in merging groups; however, although the potential accumulation of error was deemed acceptable for the purposes of the coarse stage (discussed in Section 4.3.4), it is neither desirable nor unavoidable at the fine stage.

Rather than relying on compositions across an arbitrary number of hops as required by coarse grouping, a pair of nodes attempting to determine their relative pose can now communicate directly to find the shortest path along the existing pairwise fine pose estimates and thus obtain a composition with a minimum of error.[6] A node $A$ may find such an estimate $P_{AB}$ relative to a node $B$ according to the following algorithm (suppose $FP_A$ represents the set of fine pose estimates at node $A$):

1. If $P_{AB} \in FP_A$, select $P_{AB}$ and end.
2. For each $P_{AX} \in FP_A$, request $FP_X$ from node $X$. If $P_{XB} \in FP_X$, select $P_{AB} = P_{AX} \circ P_{XB}$ and end.
3. For each $P_{XY} \in FP_X$, request $FP_Y$ from node $Y$. If $P_{YB} \in FP_Y$, select $P_{AB} = P_{AX} \circ P_{XY} \circ P_{YB}$ and end.

[6] It may be more accurate to explicitly store the actual fine registration error values with each fine pose estimate and scale the "length" of each hop (path segment) by this value. For simplicity, in this work, it is assumed that fine pose estimates meeting the error criterion are roughly equivalent, and thus each hop is considered to have unit length when determining the shortest path.

4. Continue until $P_{AB}$ has been found.

As indirect fine pose estimates are found (even intermediate ones that were not requested) they should be added to $FP$, to avoid unnecessary repetition of network requests and computations. Depending on how queries to the network are designed, these poses could be calculated on an as-needed basis, or a final step could be added to the calibration algorithm to force each node to build $FP$ to include every other node in the network. In a large network, it might also be advantageous to intelligently order this process in order to minimize communication and computation time.

### 4.3.6 Distributed Operation

For this algorithm to be truly applicable to distributed smart stereo camera networks, it must be capable of operating without establishing network-wide synchronization. This paradigm also creates fundamental future allowances for node failure, communication delays and outages, and dynamic calibration.

A scheme which lends itself particularly well to this problem is the asynchronous passing of *messages* between *processes*. According to previous assumptions and requirements, the following three commonly-used assumptions about message passing [57] are made here:

1. Transmission is made without any duplication of messages.
2. Transmission is made without any change to the messages.
3. The delay in delivering a message, though random, is finite (no loss of messages).

In keeping with the homogeneity requirement, it is desirable that the processes be *textually symmetric* at the node level. However, it is clear from the two-stage split, described in Section 4.3.2, that the algorithm involves at least two processes at each node, which are asymmetric. Therefore, a somewhat unconventional distributed algorithm concept is employed, which can be observed on two levels. At the *local* level, a number of asymmetric processes on a single node share local state information and other local resources, but are otherwise fully in the global scope and interact directly with processes on other nodes. At the *global* level, a number of nodes contain a textually symmetric *set* of processes, which do not share state information or any other resources and communicate only through messages. It is important to note that interactions between processes local to the same node operate on the *global* level; the processes simply happen to access and update the same local state information.

The dissemination of features for coarse matching and the initiation of pairwise pose refinement can be thought of as initiator processes similar to those described in [34], with additional processes only operating on incoming messages; however, this is again replicated across all nodes, so like the aforementioned asymmetry, the diffusing computations paradigm applies to the local structure at each node but not the algorithm as a whole.

#### Effect on Grouping

In the absence of network-wide synchronization, the coarse grouping procedure described in Section 4.3.4 becomes somewhat more complicated. Messages related to group merging and updates being received out of order and from multiple sources would be impossible to reconcile at any individual node. To rectify this problem, two mechanisms are established:

1. Only the current leader of a group has the authority to modify group parameters and composition.
2. Two group leaders must lock out all other changes and synchronize to each other for the duration of a merge.

Non-leader nodes forward merging information up to their group leaders (even messages originally destined for them when they were the group leader). Group leaders sort out the details and decide how to proceed, often rejecting outdated requests. In turn, non-leader nodes act only on authoritative group updates from their current leaders, sometimes deferring more recent updates pending completion of previous ones. Section 4.4.2 details how this is implemented.

## 4.4 Distributed Calibration Algorithm

The algorithm is split into ten different processes at each node; six for coarse grouping, and four for pairwise pose refinement. These processes execute within the context of their respective nodes' data spaces. Each process acts upon receipt of a message, with the exception of the *feature selection process*, which executes periodically, and the *pose refinement initiator process*, which executes whenever the group composition is updated. Termination of each process occurs based on various conditions (as not all processes necessarily operate throughout the entire calibration), and calibration is complete at each node when all processes at that node have terminated.

There are four parameters intrinsic to the algorithm itself, which have been described at length in Section 4.3: the feature size $f$, the similarity threshold $t_d$, the match threshold $t_m$, and the consistency threshold $t_c$. Certain other implementation-specific parameters are also required, notably those for the coarse and fine registration algorithms; in particular, $t_{ec}$ and $t_{ef}$ are referenced here as generic error thresholds for the coarse and fine registration algorithms, respectively. All such parameters are, of course, expected to be symmetric across all nodes, as part of their textual symmetry.

### 4.4.1 Initialization

#### Self Initialization

Initialize **nodeid**, the unique identifier (node ID) for this node, and set **groupid** (this node's current group) to **nodeid**, putting this node in its own singleton group.

Initialize two associative arrays, **coarsepose** and **finepose**, indexed by node ID. These arrays store the coarse and fine pose estimates, respectively, relative to the node ID in the key.[7] Set the **nodeid** (self) index for both to the identity pose $P_I$.

Initialize the *merge lock*, a simple node-local lock synchronization mechanism [57] which any process can acquire and which blocks any subsequent process attempting to acquire it until the current process has released it. Initialize the *group update event*, a flag which any process can set, check, or reset asynchronously.

#### Network Initialization

As per the abstract network assumption (Section 4.2.5), this node is assumed to have routing information for all other nodes in the network. Initialize any data structures necessary to perform routing of messages. Particularly, initialize a distribution of nodes for feature categorization in the geometric descriptor space, such that a node ID is returned for any descriptor value (this shall be referred to as the *binning function*).

#### Point Set Initialization

As per the assumption of inherent stereo 3D vision in Section 4.2.6, this node is assumed to have the ability to detect a reasonably repeatable set of interest points from the environment by stereo triangulation. Initialize an array **points** to contain a set of 3-tuples representing the triangulated positions of the points within this node's local coordinate system.

#### Process Initialization

Start all processes described in Sections 4.4.2 and 4.4.3.

### 4.4.2 Coarse Grouping

#### Feature Selection Process

Periodically, populate an array **feature** with $f$ different elements from **points**, either randomly or according to another selection model. Assign it a locally unique **featureid**, such as a sequential number. Compute the geometric descriptor of **feature** (as **descriptor**), and select the destination matching node according to the binning function. Send the output message to the *feature matching process* on the destination node.

- **Output:** nodeid, featureid, descriptor, feature

[7] One might expect that a node would only need to store its coarse pose estimate relative to its current group leader, since it may only be a member of one group at a time, and thus conclude that storing estimates from previous groups is unnecessary. This is fundamentally correct reasoning, but the asynchronicity of the processes which access the coarse pose estimate means that negotiations already in progress but not completed when the current estimate changes still rely on these previous estimates, and it is more efficient to store them locally at each node than to embed them in messages.

### Feature Matching Process

- **Input:** sourceid, featureid, descriptor, feature

This process maintains an array **matchdb** which contains all previously received features.

Compare the incoming **feature** (as the data) to each feature stored in **matchdb** (as the model) to which it meets the similarity condition (where the difference between their **descriptor** values is less than $t_d$) through the coarse registration algorithm. If the source node of the new feature is $S$ and the source node of the matching database feature is $M$, the pose returned will be $P_{SM}$. For each case where the final registration error $e < t_{ec}$, perform an unbiased, deterministic node selection[8] between $S$ and $M$. Send one of the following messages to the winning node's *match processing process* depending on which node is selected:

- A message to $S$, where **otherid** is $M$ and **cpose** is $P_{SM}$.
- A message to $M$, where **otherid** is $S$ and **cpose** is $P_{MS} = P_{SM}^{-1}$.

Finally, add the incoming feature (along with its source node ID, feature ID, and geometric descriptor) to **matchdb**.

- **Output:** otherid, featureid, cpose

### Match Processing Process

- **Input:** otherid, featureid, cpose

This thread maintains an associative array **matches**, indexed by node ID, each entry containing an array of feature IDs and relative pose estimates returned by coarse matching between this node and the other node.

Ensure that **otherid** is not already in this node's group, that **otherid** is not marked as complete in **matches**, and that **matches:otherid** does not already contain a match with this **featureid**. Add the **featureid** and **cpose** to **matches:otherid**.

If there are now at least $t_m$ matches to node **otherid**, for each unique combination of $t_m$ matches which includes the incoming match:

1. Calculate the average pose transformation of the poses associated with the matches.
2. If $||P_m(p_c) - P_{avg}(p_c)|| \leq t_c$ for every match with associated pose $P_m$:
   (a) Send the output message to the *group merge initiator process* of the current group leader, where **cpose** contains this node's current group coarse pose estimate and **apose** contains $P_{avg}$.
   (b) Mark this node as complete in **matches**.

[8] This node selection function *must* be deterministic in the sense that it returns the same node when executed on a given pair of nodes no matter which matching node is performing the selection, so that matches between that pair of nodes are always routed to the same node in the pair, and *should* be as unbiased as possible so that some nodes do not tend to receive more matches for processing than others.

(c) Exit the loop.

3. Continue with the next unique combination.

- **Output:** nodeid, otherid, cpose, apose

### Group Merge Initiator Process

- **Input:** sourceid, otherid, cpose, apose

While this node is the group leader, if **sourceid** is not in this node's group, attempt to acquire the *merge lock*.

If this node is no longer the current group leader[9] and **otherid** is not already in this group, forward the input message to the current group leader.

Otherwise, initiate a merge with **otherid** by sending a merge output message to its *group merge responder process*, where **group** is an array containing all node IDs in this group and **cpose** is initialized to $P_I$.

- **Output:** nodeid, group, cpose

Wait for its acknowledgement message.

- **Input:** sourceid, otherid, opose

If the acknowledgement message is a preemption signal (see the *group merge responder process* description), reinsert the pose input message into the queue and return to message processing. Otherwise, based on this message, construct a group update message, where **newgroupid** is the **sourceid** in the message, **ogroup** and **opose** are its returned group contents and relative pose change (see the *group merge responder process* description) respectively, and **cpose** and **apose** are obtained from the original input message. Send this update message to the *group update process* of each other member of this group. Update this node's coarse pose estimate relative to **sourceid** (its new group leader) according to Equation 4.10. Release the *merge lock* and set the *group update event*.

- **Output:** nodeid, newgroupid, ogroup, opose, cpose, apose

Once this node is no longer its group leader, forward all remaining messages destined for this process to the current group leader.

### Group Merge Responder Process

- **Input:** sourceid, ogroup, cpose

[9]Due to the asynchronicity of message passing in this scheme, the situation might easily arise where a node's match processing process sends its output to the current group leader, and a subsequent merge changes the group leader before the message is actually delivered or before this process successfully acquires the merge lock.

While this node is the group leader, attempt to acquire the *merge lock*. If this process does not acquire the lock for a certain random interval (within a specified range), assume that a merge deadlock has occurred and send a special acknowledgement message to this node's *group merge initiator process* containing a signal that this process wishes to preempt the other. Once the *merge lock* has been acquired, ensure that the initiating source node is not in this node's group before proceeding.

If this node is no longer the current group leader, compose this node's current group coarse pose estimate into the input message's **cpose** and forward the message to the current group leader.

Otherwise, acknowledge the merge with **sourceid** by sending an acknowledgement message to its *group merge initiator process*, where **cpose** is repeated from the input message (note that any necessary changes in this pose will have been incorporated by the pose compositions included in the leader forwarding).

- **Output:** nodeid, group, cpose

Construct a group update message, where **ogroup** is repeated from the input message and the three pose entries (**opose**, **cpose**, and **apose**) are all set to $P_I$. Send this update message to the *group update process* of each other member of this group. Set the *group update event*.

- **Output:** nodeid, newgroupid, ogroup, opose, cpose, apose

Once this node is no longer its group leader, for all remaining messages destined for this process, compose this node's current group coarse pose estimate into the message's **cpose** and forward the message to the current group leader.

#### Group Update Process

- **Input:** sourceid, newgroupid, ogroup, opose, cpose, apose

Since group update messages may arrive out of order but cannot be processed this way, this process differs from the others in that it waits specifically for a message from its current group leader. This ensures consistency, as discussed in Section 4.3.6.

Set this node's **groupid** to **newgroupid**, update its coarse pose estimate according to Equation 4.11 using **opose**, **cpose**, and **apose**, and append the nodes in **ogroup** to this node's group list.

### 4.4.3 Pairwise Pose Refinement

#### Pose Refinement Initiator Process

Wait for the *group update event* to be set. Reset the *group update event*. For each new node in the group, perform an unbiased, deterministic node selection between it and this node. Send an initiation message to the *responder process* of each node which wins the selection and for which this node does not already have an entry in **finepose**, where **cpose** is this node's current group coarse pose estimate.

- **Output:** nodeid, groupid, cpose

**Pose Refinement Responder Process**

- **Input:** sourceid, sgroupid, spose

If this node does not have an entry for **sourceid** in **coarsepose**, continue checking each time the *group update event* is set until it does.

Compute a relative pose estimate between this node and the source node based on **spose** and this node's coarse pose estimate relative to **sgroupid**. From this, determine a cone approximation of the source node's field of view (see Figure 4.5) within this node's local coordinate system. Populate an array **fpoints** with all points in **points** falling within this cone. If **fpoints** contains at least 3 points, respond to the source node's *registration process* with the output message where **cpose** is this node's coarse pose estimate relative to **sgroupid**.

- **Output:** nodeid, sgroupid, cpose, fpoints

**Fine Registration Process**

- **Input:** sourceid, sgroupid, spose, spoints

Similarly to the *responder process*, compute a relative pose estimate, determine a field of view cone, and populate an array **fpoints**. If **fpoints** contains at least 3 points, attempt fine registration on **spoints** (as the data) and **fpoints** (as the model). If the registration error $e < t_{ef}$, set this node's entry for **sourceid** in **finepose** to the result, and send the output message to the source node's *update process*, where *rpose* is the inverse of the result.

- **Output:** nodeid, rpose

**Pose Update Process**

- **Input:** sourceid, rpose

Set this node's entry for **sourceid** in **finepose** to **rpose**.

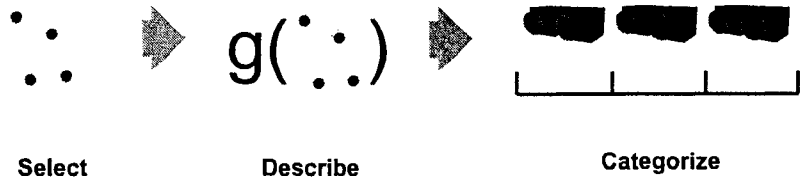


Figure 4.6: Feature Selection Process

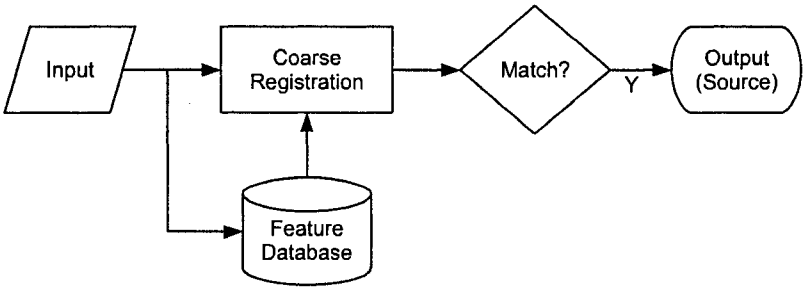


Figure 4.7: Feature Matching Process

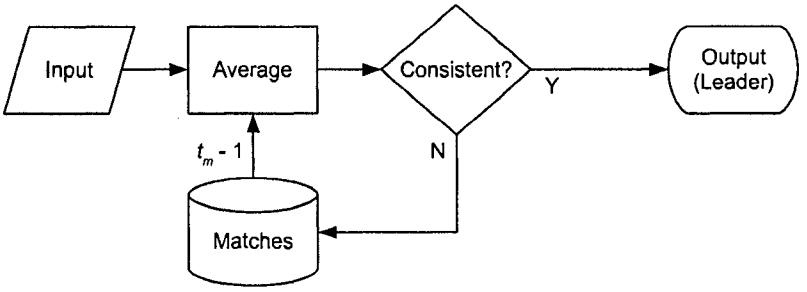


Figure 4.8: Match Processing Process

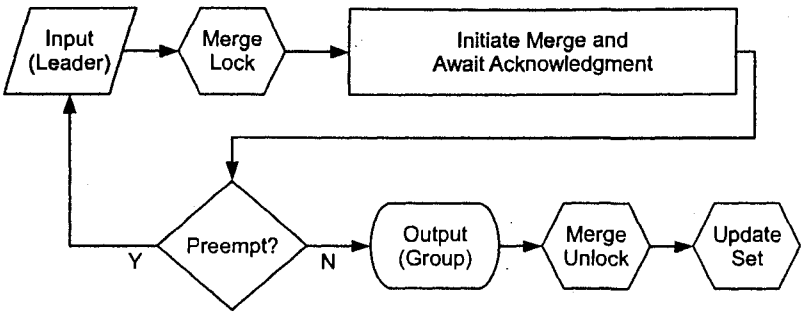


Figure 4.9: Group Merge Initiator Process

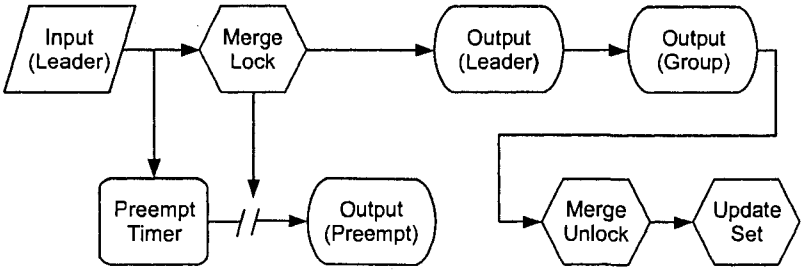


Figure 4.10: Group Merge Responder Process

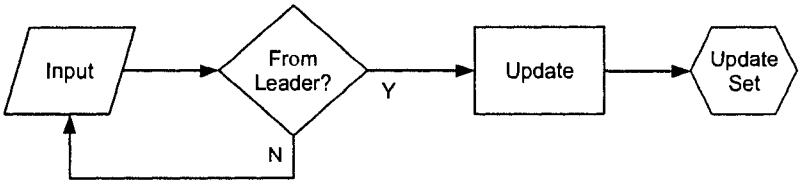


Figure 4.11: Group Update Process


Figure 4.12: Pose Refinement Initiator Process

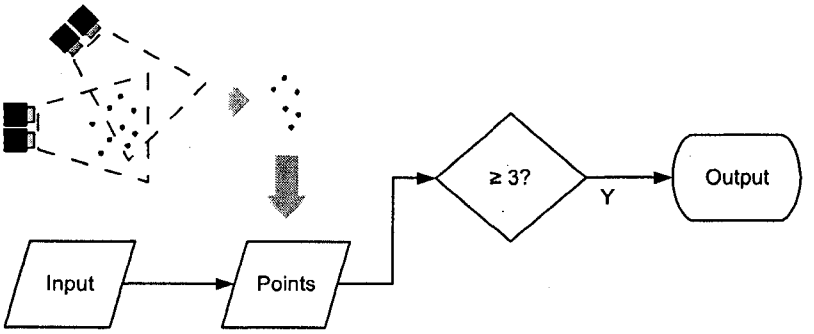


Figure 4.13: Pose Refinement Responder Process

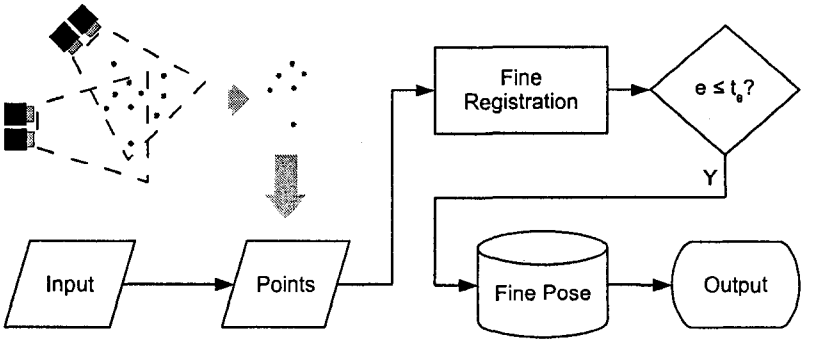


Figure 4.14: Fine Registration Process

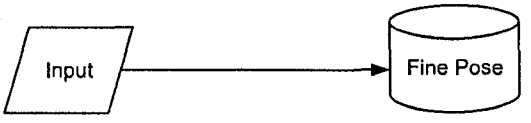


Figure 4.15: Pose Update Process

# Chapter 5

# Implementation Details

## 5.1 Stereo 3D Vision

### 5.1.1 Calibration

Camera calibration (both of individual cameras and of stereo pairs) is generally a fastidious process. Many stereo camera calibration methods attempt to reduce the complexity of the parameters or the level of interaction required to perform calibration, at the expense of precision. In most applications, the time and interaction required for full calibration is impractical or impossible, so these methods provide a feasible alternative when a large degree of imprecision can be tolerated. In the case of distributed smart stereo cameras, however, precision is vital for proper matching and reconstruction across nodes.

The pre-deployment offline access assumption (Section 4.2.1) allows for stereo camera calibration, among other things, to be performed in an offline setting with full control over the environment. Therefore, it is desirable to employ a calibration method whose parameters will yield the highest possible precision when performing 3D reconstruction (in this case, point triangulation), regardless of the time and interaction level required to achieve it.

The Camera Calibration Toolbox for Matlab [64] implements these methods with an interactive interface. This software is used to initially calibrate the physical camera nodes for the experiments in Chapter 6.

### 5.1.2 Correspondence

Assuming that suitable interest points can be detected in both images – which will be realized in Section 5.2 – in order to determine the 3D positions of the physical points, the image points must be brought into correspondence with one another.

The zero-mean normalized cross-correlation (ZNCC) score is a measure of similarity for template matching, frequently used (among others) in computer vision for pixel correspondence. In this case, it is used to compute the correlation between interest points in each image. Equation 5.1 is an expression for the ZNCC

score between a window $L_c$ of size $W \times W$ about an interest point in the left image and a window $R_c$ of equal size about one in the right image:

$$ZNCC = \frac{\sum_{j=1}^{W}\sum_{i=1}^{W}[R_c(i,j)-\mu(R_c)]\cdot[L_c(i,j)-\mu(L_c)]}{\sqrt{\sum_{j=1}^{W}\sum_{i=1}^{W}[R_c(i,j)-\mu(R_c)]^2}\cdot\sqrt{\sum_{j=1}^{W}\sum_{i=1}^{W}[L_c(i,j)-\mu(L_c)]^2}} \tag{5.1}$$

Of immense benefit to the accuracy and speed of correspondence is the existing knowledge, from the calibration in Section 5.1.1, of the camera pair's intrinsic and extrinsic parameters. This allows for the application of the epipolar constraint [33] to correspondence. For each interest point in the left image for which a correspondence in the right image is sought, the right-image epipolar line can be calculated according to the following procedure:

1. Normalize the pixel coordinates of the left-image interest point based on the left camera intrinsics.
2. Calculate the right-image epipolar line (in normalized image coordinates) for this point based on the stereo extrinsics.
3. Transform the normalized epipolar line into right-image pixel coordinates based on the right camera intrinsics.

Only right-image interest points falling within a certain distance of this epipolar line[1] need be considered as potential correspondences, greatly reducing the probability of false matches and the number of ZNCC score computations required.

### 5.1.3 Point Triangulation

An algorithm for triangulating the position of a 3D point based on the normalized image coordinates of its respective projections in a stereo image pair is described in [33]. It derives two expressions for estimating the 3D coordinates. Ideally, these are equal, but in practice, image noise and error in camera calibration cause them to differ. In this case they are averaged to yield a more accurate estimate.

## 5.2 Interest Point Detection

The stable detection of interest points, reviewed in Section 2.2, is fundamental to this work. The effectiveness of the method used determines how well the 3D point sets correspond across nodes with different viewpoints, which has a huge impact on registration performance and thus the performance of the overall calibration algorithm.

[1] Since the calibration procedure in Section 5.1.1 includes optical distortion, the epipolar line, when drawn in the right-image pixel coordinate space, will in fact be a curve, generally speaking.

### 5.2.1 Requirements

The requirements for interest point detection performance are imposed from three sources: the correspondence algorithm, the coarse matching scheme, and the fine registration algorithm. The requirements can be described by two quantitative performance criteria of the interest point detector: the total number of one-to-one correspondences between a pair of images or nodes, and the *repeatability* of the points detected. The repeatability score for interest point detection between a given pair of images or nodes is quantitatively calculated as the ratio of the number of one-to-one correspondences to the (minimum) total number of detected points [19]. These two criteria affect the performance of each of the three algorithms differently.

#### Correspondence

The correspondence algorithm of Section 5.1.2 primarily responds to repeatability. However, this only applies to stereo images locally at each node. Since finding points for stereo correspondence is one of the primary applications of interest point detection, most methods will perform suitably. The main challenges for repeatability are changes in scale (focal length) and motion (point of view) between the two images, neither of which are significant enough in small-baseline stereo to have a serious effect. Regardless, however, if the interest point detector cannot provide a high repeatability rate between the stereo images at one node, it is highly unlikely that the resultant triangulated points will exhibit satisfactory repeatability for the registration stages.

#### Coarse Matching

The coarse matching scheme described in detail in Section 4.4.2 has a special performance requirement dependent on both the number of correspondences and the repeatability score between two nodes attempting to match features.

As explained in Section 4.3.3, the performance of the matching scheme depends on the probability of matching two randomly selected features between two given nodes. In order to relate Equation 4.5 to the performance criteria of the interest point detector, the criteria must be expressed in the terms used in that section. Equations 5.2 and 5.3 describe the number of correspondences $N$ and the repeatability score $R$, respectively.

$$N = |S_A \cap S_B| \tag{5.2}$$

$$R = \frac{|S_A \cap S_B|}{\min(|S_A|, |S_B|)} \tag{5.3}$$

It is clear from Equation 4.5 that $P(F_A \approx F_B)$ increases if $|S_A \cap S_B|$ increases relative to $|S_A|$ and $|S_B|$; this translates into a desired increase in $R$.

Additionally, it is required that $N$ be large enough to actually contain at least $t_m$ features, imposing the constraint $C(N, f) \geq t_m$. It should be mentioned that, as explained in Section 4.3.3, optimal performance

is achieved with a minimum $N$ satisfying this inequality, for a given $f$. However, in accordance with the homogeneity requirement, all nodes must use the same parameters for interest point detection, and there is no *a priori* way to know $N$ for a given pair of nodes, so the detector must yield a fairly large value of $N$ to ensure that the minimum requirement is satisfied in the vast majority of applicable pairwise cases.

#### Fine Registration

In Section 5.3.2, it is explained that this implementation uses the Trimmed Iterative Closest Point (TrICP) algorithm [15] for fine registration. This registration method is applicable to point cloud overlaps under 50%. As it will likely begin with a relatively good initial alignment from the coarse pose estimation result, but also considering occlusion effects and error in field of view estimation, 50% shall be considered the minimum repeatability criterion for good performance. Also, in general, the TrICP algorithm's performance increases as $N$ increases.

There are two qualifications to be noted here. The first is that, strictly speaking, the repeatability must occur between the 3D point sets at each node, which does not necessarily translate directly into repeatability between any particular pair of 2D images from those nodes. However, given the correspondence method chosen in Section 5.1.2, it can be considered essentially identical to the repeatability in detecting points in the nodes' respective left images. The second is that the repeatability need only be measured across images with nearly identical scene contents (occlusion notwithstanding), as before fine registration takes place, only the points falling in the estimated intersection of the nodes' fields of view are selected. The actual point of view, however, may vary arbitrarily under the current set of assumptions.

#### Summary

Based on the individual requirements, the following criteria are applied to interest point detection:

$$C(N,f) \geq t_m \tag{5.4}$$

$$R \geq 0.5 \tag{5.5}$$

It should be stressed that Equation 5.5 is not an absolute requirement – the calibration algorithm may still converge if it is not met. In any case, neither of these criteria can be directly applied to the selection of an interest point detection algorithm. Rather, they are intended to simultaneously guide the selection of the interest point algorithm, the imposition of deployment constraints, and the extent of scene control.

### 5.2.2 Algorithm

Ideally, an interest point detection algorithm would be chosen to meet the requirements in Equations 5.4 and 5.5 without imposing any deployment constraints beyond a connected vision graph and without requiring any control over the scene. However, stable interest point detection across widely separated views in 3D is a

difficult problem, and as the evaluation in [25] would suggest, none of the methods mentioned in Section 2.2 are satisfactory on their own to ensure convergence of the calibration algorithm.

For practical purposes, the FAST detector [22, 23] is selected for this implementation. Convergence can be encouraged by constraining nodes to share large portions of their fields of view or by calibrating on a scene with strong interest points. In order to meet the repeatability requirements with a manageable number of points in a real-world scenario, it is likely that control over the scene during calibration is necessary for convergence. An example is the use of objects with strong textural features in the automatic point set experiments in Section 6.4.

Though this imposes severe limitations on the general applicability of this implementation, interest point detection is still a very active research topic in computer vision, especially within the 3D context, and future improvements in stability across widely separated views will undoubtedly improve the situation. While it is impossible to ensure convergence regardless of the scene contents, it is desirable to generalize the system to as broad a range as possible so that good results are obtained in most practical cases.

## 5.3 Registration

### 5.3.1 Coarse Registration

The size $f$ of features used for matching in the coarse grouping stage of calibration is necessarily small – likely too small to allow for anything but perfect overlap of the points. An excellent solution to this registration problem is the fully-contained version of the DARCES algorithm [13], using three control points. DARCES without the RANSAC component is a relatively simple algorithm, allowing it to perform matching very quickly on a large number of features.

For this purpose, DARCES simply requires an error threshold $t_{ec}$, which dictates the maximum Euclidean distance of each successive control point from its expected location as the points are found, and of the remaining points from their expected locations.

### 5.3.2 Fine Registration

The concept of the Iterative Closest Point (ICP) algorithm [14] lends itself well to the fine registration problem encountered in pairwise pose refinement, since it directly returns a pose estimate and the registration error. However, the difficulty of stable interest point detection, occlusion effects, and uncertainty about the overlap in field of view all contribute to poor overlap in the point sets used for pose refinement, and ICP's performance degrades heavily when the point sets do not fully overlap. The Trimmed Iterative Closest Point (TrICP) algorithm [15], used in this implementation, overcomes this limitation by operating on the best subset of points, which can be automatically tuned to any degree of overlap, and is applicable to overlaps under 50%.

The TrICP algorithm requires an error threshold $t_{ef}$ and a minimum change per iteration $t_r$ as stopping conditions. Additionally, since in this case the overlap is variable, the $\zeta$ parameter must be automatically set using an objective function, which requires a weight parameter $\lambda$. Finally, a separate maximum error

threshold $e_{max}$ is required to decide whether to accept or reject the fine registration.

### 6.1.2 Accuracy

Accuracy is the measure of the error in the algorithm's resulting pose estimates.

A method of evaluating accuracy has been developed for this work. Since the calibration algorithm is based on the three-dimensional point as its data primitive, the mean error in a pose estimate can be determined by averaging the Euclidean distance between a number of points with ground-truth correspondence, detected and triangulated at the nodes separately from those used for calibration. Although error accumulates with the path length (number of pose compositions) in the calibration graph, it is more relevant to consider the length of the path in the vision graph,[1] since the 3D reconstruction consistency among nodes observing the same part of the scene is the likely criterion. The mean 1-hop error, then, is the average Euclidean distance between the ground-truth point sets pairwise between all nodes with edges on the vision graph. The mean 2-hop error considers pairwise paths up to two edges in length, and so forth until the entire vision graph is considered. How relevant multi-hop accuracy is depends on the application.

### 6.1.3 Scalability

Scalability is the measure of the effect on the algorithm's performance of the number of nodes in the network. The three primary resources to consider are node-local computing resources (i.e. CPU and memory), node-local data storage, and network bandwidth. For a given network, the addition of a node will presumably increase the consumption of each of these resources; it is desirable to minimize this increase.

In order to properly evaluate scalability, it is necessary to examine individual factors arising from the algorithm itself. The most significant of these can be summarized in terms of the number of nodes $n$ as follows:

- Feature dissemination requires bandwidth resources in $|N|$ *per node*.
- Feature matching (coarse registration) requires computing and storage resources in $|N|$.

This assumes that the vision graph maintains an approximately constant number of pairwise edges regardless of $|N|$, as would be the case with most applications. In cases where this assumption does not hold, it is necessary to add a third factor:

- Pairwise pose refinement computation (fine registration) requires computing resources in $|N|$.

Scalability in all three resources can be quantized experimentally in terms of the above factors. For node-local computing resources, each of the registration operations (coarse and fine) is assigned a weight depending on its relative computational requirements, and the total number of times these operations are executed at each node is recorded. For node-local storage resources, the final size of the matching database, in number of features, is recorded. For network-wide bandwidth resources, the outgoing bandwidth usage per second at each node is recorded.

[1] As was mentioned in Section 3.3.3, the calibration graph structure approaches that of the vision graph. It is clear that maximizing the number of edges, thus minimizing the number of hops between nodes with edges on the vision graph, is beneficial to the accuracy metric.

The actual effect of increasing resource consumption differs depending on which is most significant. The convergence time of the calibration algorithm depends either on the total bandwidth usage as it relates to network capacity, or on the total node-local processing as it relates to throughput on each node. The ability of the calibration algorithm to complete at all may be affected by the storage use as it relates to the available storage on each node.

## 6.2 Equipment and Software

### 6.2.1 Stereo Cameras

Four physical stereo camera rigs with adjustable mounts were constructed for experimentation (Figure 6.1. Each consists of two Prosilica EC1350 1.4 megapixel digital CCD IEEE-1394 cameras with Computar M3Z1228-MP manual focus lenses. The aluminum mounting frames allow adjustment and fixation of the relative $x$-axis translation (baseline) and $y$-axis rotation between the cameras. After calibration, these rigs are tested as having a mean triangulation error within their range of 2.58%, translating to approximately 0.5 to 2.0 millimetres depending on the distance in $z$.

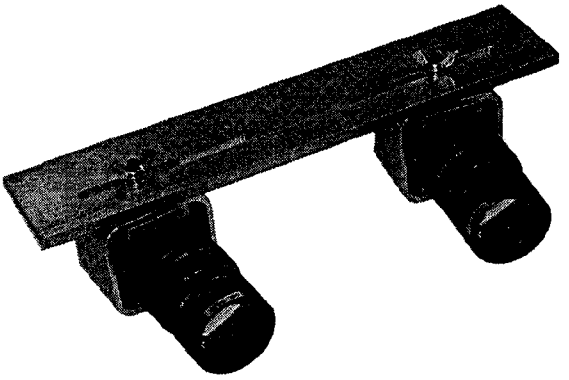

Figure 6.1: Adjustable Stereo Camera Mount (Node)

All four rigs are connected via a National Instruments IEEE-1394 image acquisition controller to a single Intel Pentium D (x86_64) PC. This is strictly for practical efficiency: no centralized processing of any kind is performed, and the software is designed and tested to work with each rig connected to its own PC (effectively forming a "node") communicating over a TCP/IP network; this PC, however, has enough resources to run four nodes at once, with communication on local TCP/IP.

For the purposes of experiments using these camera rigs, the associated nodes are named Baureo, Lirr, Mayestril and Sheerek, sometimes abbreviated as B, L, M and S, respectively.

### 6.2.2 Distributed Calibration Software

The main distributed calibration algorithm is implemented as a single multi-threaded program for each node, with a main initialization and message passing program overseeing the execution of ten other threads (one for each process in Section 4.4). It reports verbose time-stamped status messages so that the progress of calibration can be observed, and offers an interactive shell with various options for viewing and storing the results once calibration is complete; these would, of course, be unnecessary and therefore disabled on real embedded nodes.

The calibration software relies on external files as input for network information and the point sets for each node. The network description file, in this case, is generated manually, as it represents the information that would normally be obtained at a previous stage of routing initialization in an ad-hoc network. The point files can be generated by the local point detection software, similar to the expected situation in a real distributed smart stereo camera network with an embedded node architecture, or by other means for experimentation.

The distributed calibration software is implemented in Python [65]. See Section B.1 for full annotated source code.

### 6.2.3 Local Point Detection Software

Representing the initialization stage of calibration, the local point detection is actually implemented as a separate piece of software from the rest of the calibration algorithm for the purposes of experimentation. This division allows for manual control of the detection stage, examination and manipulation of its results, and, in the case of the virtual point experiments described in Section 6.5, complete removal of vision-based point detection.

The software presents an interface for entering and storing local stereo camera calibration parameters. These are used to capture images, detect image interest points (either manually or using the FAST detector), perform automatic correspondence on the points if necessary, and triangulate the 3D coordinates of the associated points. The points are then saved to a file which can be taken as input by the distributed calibration software. Figure 6.2 shows the graphical interface to the local point detection software.

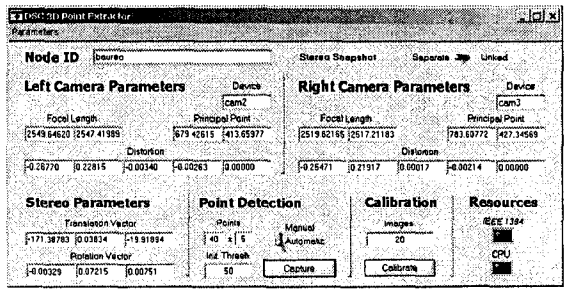


Figure 6.2: Local Point Detection Software GUI

Under automatic operation, once point detection is complete, the distributed calibration software de-

scribed in the previous section is launched externally for the node. In this way, the full calibration process occurs completely automatically after a single button click in the GUI for each node, which is considered functionally equivalent to signalling a self-contained embedded device to begin calibration by some other means.

The software performs simultaneous dual image acquisition, interest point detection in each image, correspondence of the interest points between the images, and triangulation of the 3D coordinates of each point in the camera coordinate system. Thus, it is able to directly produce the requisite 3D point set for a node. Although this is designed to operate fully automatically, there are various options for user verification and intervention. Notably, it is possible to override automatic interest point detection and correspondence with a series of manual mouse clicks in each image.

The local point detection software is implemented in ANSI C, using the National Instruments LabWindows/CVI development environment and vision libraries for image acquisition and manipulation. See Section B.2 for full annotated source code.

## 6.3 Manual Point Set

In order to test the capabilities of the calibration algorithm and tune its parameters under controlled conditions, the first experiment series is designed to operate on manually selected points with full correspondences across all four nodes. The primary purpose of this experiment type, once suitable parameters are found, is to test the effects of different point set sizes and overlap characteristics on convergence and accuracy.

### 6.3.1 Apparatus

The four stereo camera rigs are mounted within the vision platform such that each has full view of the experimental target. On each node, the local point detection software is run with manual point detection enabled, and a predetermined ordered series of 100 points on the target is selected. Four sets of 100 3D points are thus produced, with any given index in one node's point set corresponding to the same physical point represented by that index in the other three nodes' point sets.

#### Point Set Size and Overlap

With a large controlled point set for each node, it is possible to extract a number of subsets of different sizes and with different relative overlaps. In this experiment series, overlap is *staggered* to approximate the situation where nodes share only part of their field of view. This has a dual effect. Obviously, it will challenge the coarse grouping portion of the calibration algorithm proportionately to the overlap ratio. More subtly, however, since in reality the nodes all mutually share their fields of view, they will share their whole point sets for fine pairwise pose estimation, thus testing that portion's performance for the same overlap ratio.

### 6.3.2 Procedure

A total of 22 point subsets are extracted from the data, and each is tested using the distributed calibration software, with all four nodes running locally on the same workstation.[2] This procedure is repeated twice for each subset, and the average results for convergence time and mean error are calculated and recorded.

#### Subset Extraction

Point subsets $B_{n,p}$, $L_{n,p}$, $M_{n,p}$ and $S_{n,p}$, of size $n$ and overlap ratio $p$, are extracted from the original point sets $B$, $L$, $M$, and $S$ according to the following algorithm (assuming zero-indexed sets):

1. Calculate the number of non-overlapping points $e = n - np$, and the total pool size $P = n + 3e$.
2. Iterate $i$ from zero to the pool size $P$:
   (a) Select a unique random integer value $0 \leq r < P$.
   (b) If $i < n$, add $B_r$ to $B_{n,p}$.
   (c) If $e \leq i < n + e$, add $L_r$ to $L_{n,p}$.
   (d) If $2e \leq i < n + 2e$, add $M_r$ to $M_{n,p}$.
   (e) If $i \geq 3e$, add $S_r$ to $S_{n,p}$.

Point set sizes of 20, 30, 40 and 50 were extracted, with the largest pairwise overlap ranging between 50% and 100% in increments of 10%. Note that the pool size required for overlaps of 50% and 60% with 50 points is larger than 100, so these combinations are not tested.

#### Calibration Parameters

The matching parameters to the calibration algorithm are chosen as $f = 4$ and $t_m = 3$. Based some testing of the accuracy of the cameras and their stereo parameters, the other two main parameters are set as $t_d = 10.0$ and $t_c = 5.0$. The cone approximation of the field of view extends 3.0 metres from the focal point, at an angle of $\pi/3$ from the focal axis.

Again based on camera accuracy, some parameters are chosen for the registration algorithms. The coarse error threshold is set as $t_{ec} = 2.8$. The fine error threshold is set as $t_{ef} = 1.0$, with aminimum change of $t_r = 0.01$ per iteration, an objective function weight $\lambda = 2.0$ minimizing the error for $\zeta$ in the range $[0.4, 1.0]$ (see the automatic overlap setting procedure in [15]), and a final maximum mean squared registration error of $e_{max} = 100.0$.

Based on the network and processor capabilities of the host PC, feature dissemination is set to occur every $0.08 + 0.0001i$ seconds, where $i$ is the number of features sent from the node.

[2] AMD Athlon 64 3700+ (2.2 GHz), 1536MB RAM, running Gentoo Linux (kernel version 2.6.24) and Python 2.4.

#### Convergence Time

The convergence time is recorded from the start of initialization to the completion of all threads. Note that the convergence time is primarily valid as a relative measure of execution time; as is discussed in Section B.1, the absolute performance of the algorithm could be greatly improved by reimplementation in a compiled language executing locally on smart camera devices, a subject for future work.

#### Mean Error

The mean error is calculated between all pairs of nodes as described in Section 6.1.2. For each permutation $N_i$,$N_j$ of the node set $N$, the fine pose estimate $P_{ji}$, which maps the points of $N_j$ to the coordinate system of $N_i$, is obtained (if it is available directly) or computed (if it is only available indirectly; see Section 4.3.5 for details). The mean error is computed over all 100 original points by averaging the Euclidean distance of a given point in the $N_i$ set with the mapping $P_{ji}$ of the corresponding point in the $N_j$ set.

### 6.3.3 Results

The average recorded results (convergence time and mean error) of the manual point set experiments are shown in Table 6.1.

Table 6.1: Manual Point Set Experiment Results

| Points ($n$) | Overlap ($p$) | Convergence Time (s) | Mean Error (mm) |
|---|---|---|---|
| 20 | 50% | 262 | 2.6158 |
| 20 | 60% | 49 | 2.5652 |
| 20 | 70% | 54 | 2.2460 |
| 20 | 80% | 27 | 2.2841 |
| 20 | 90% | 33 | 2.1925 |
| 20 | 100% | 18 | 2.1299 |
| 30 | 50% | 371 | 2.1220 |
| 30 | 60% | 209 | 2.1438 |
| 30 | 70% | 114 | 2.0830 |
| 30 | 80% | 74 | 2.0474 |
| 30 | 90% | 48 | 2.0915 |
| 30 | 100% | 38 | 2.0582 |
| 40 | 50% | 911 | 2.0888 |
| 40 | 60% | 441 | 2.1262 |
| 40 | 70% | 276 | 2.0506 |
| 40 | 80% | 117 | 2.1257 |
| 40 | 90% | 128 | 2.0502 |
| 40 | 100% | 79 | 2.0141 |
| 50 | 70% | 271 | 2.1048 |
| 50 | 80% | 248 | 2.1025 |
| 50 | 90% | 177 | 1.9908 |
| 50 | 100% | 230 | 1.9989 |

### Convergence

All 22 point subsets converge in both runs. Figure 6.3 shows a clear trend: convergence time increases exponentially as the point set size $n$ increases or the overlap ratio $p$ decreases. This is in accordance with the theoretical probability of finding matches, as treated in Section 4.3.3, where $n = |A| = |B|$ and $np = |A \cap B|$; as $n$ increases or $p$ decreases, matches found via feature dissemination and coarse matching become more scarce in relation to the number of features disseminated (and thus, the execution time and bandwidth).

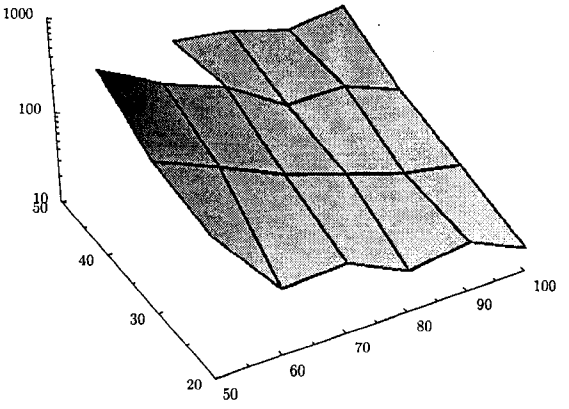


Figure 6.3: Convergence Time Trends in $n$ and $p$

### Accuracy

The resultant fine pose estimates do not vary greatly in accuracy, as pairwise estimates with large registration error are rejected and estimates for those pairs are instead calculated indirectly by composing intermediate estimates (see Section 4.3.5). This shows that the algorithm is quite robust against arbitrarily low pairwise overlap ratios even in nodes sharing large portions of their field of view, which is particularly important in cases where parts of the scene are occluded.

The general trend is that accuracy improves slightly both with point set size and overlap ratio. There is a more drastic drop in accuracy when the overlap size $np$, which is the number of points the fine registration algorithm will ultimately optimize on, is very low (this can be seen here in the lowest overlap ratios for $n = 20$).

### Interpretation

Based on these results, it can be concluded that small, robust point sets are desired at each node. However, the point set size must be large enough to provide good overlap (both in ratio and total size) with at least one other node, so that fine pose estimation has a substantial number of points to work with and yields accurate results. In this implementation, a point set size of 30 to 50 points appears reasonable, and it should be ensured

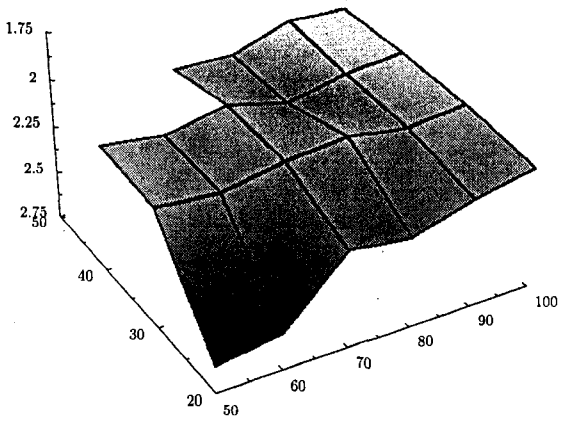


Figure 6.4: Accuracy Trends in $n$ and $p$

that the interest point detection is robust enough to overlap at least 50% within the shared field of view with at least one other node for a given deployment.

## 6.4 Automatic Point Set

Having established some general criteria for reasonably timely convergence in the manual point set experiments, the next step is to test real automatic calibration of the network. The purpose of these experiments is to test the convergence and accuracy performance of the algorithm in real conditions.

### 6.4.1 Apparatus

The four stereo camera rigs are mounted within the vision platform in a variety of configurations, all meeting the requirement that the vision graph be connected.

#### Practical Considerations

Due to the limitations of interest point detection, the current implementation cannot reliably converge calibration with only the shared view deployment constraint. To encourage nodes to detect point sets of reasonable size meeting the criteria of Equations 5.4 and 5.5, for the purposes of this series of experiments, calibration is performed on scenes with well-defined, non-ambiguous interest points. The lack of a large number of extraneous or ambiguous points in the vision platform provides part of this, but the main factor is the use of scene objects with strong textural features.

For the four automatic point set experiments conducted here, the camera rigs are placed such that their vision graph is complete, and a conical calibration target with a number of black shapes on a white surface

is placed within their shared field of view. The target has texture around its entire surface, and provides occlusion.

### 6.4.2 Procedure

Four instances of the local point detection software, configured to execute the distributed calibration software on completion, are run in automatic mode on the vision platform workstation. Convergence time and the final calibration graph are recorded. A ground truth point set is manually selected for each camera rig, and the mean error is calculated and recorded.

#### Calibration Parameters

Since the equipment configuration used in these experiments is the same as with the manual point set, the calibration parameters used are identical (see Section 6.3.2).

#### Convergence Time

As with the manual point set experiments, the convergence time is recorded from the start of initialization to the completion of all threads.

#### Mean Error

The mean error is calculated between all pairs of nodes for which pairwise pose estimates can be determined as described in Section 6.1.2. A new target is placed in the scene, and the same 20 physical points are manually selected and triangulated as the ground truth set. Then, as with the manual point set experiments, the pairwise pose estimates are computed as necessary and the mean error is computed over these 20 points.

### 6.4.3 Results

The mean error, convergence time, and final calibration graphs for each of the four experiments are shown here.

#### Experiment 1

See Figure 6.14 for the deployment used in this experiment, and Figure 6.15 for a visualization of the calibration results.

- **Mean Error:** 2.7666 mm
- **Convergence Time:** 159 s

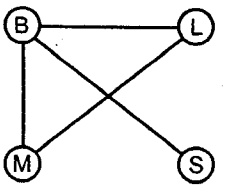


Figure 6.5: Calibration Graph for Automatic Experiment 1

**Experiment 2**

See Figure 6.16 for the deployment used in this experiment, and Figure 6.17 for a visualization of the calibration results.

- **Mean Error:** 3.0844 mm
- **Convergence Time:** 38 s

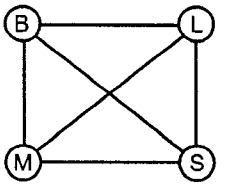


Figure 6.6: Calibration Graph for Automatic Experiment 2

**Experiment 3**

See Figure 6.18 for the deployment used in this experiment, and Figure 6.19 for a visualization of the calibration results. Note that calibration has not fully converged in this experiment, so the results apply only to the three nodes comprising the main group.

- **Mean Error:** 2.7160 mm
- **Convergence Time:** 648 s

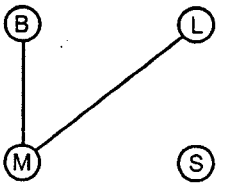


Figure 6.7: Calibration Graph for Automatic Experiment 3

### Experiment 4

See Figure 6.20 for the deployment used in this experiment, and Figure 6.21 for a visualization of the calibration results.

- **Mean Error:** 2.5813 mm
- **Convergence Time:** 240 s

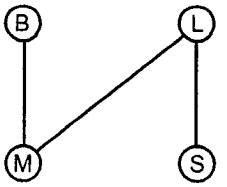


Figure 6.8: Calibration Graph for Automatic Experiment 4

#### Interpretation

The algorithm is capable in most cases of converging given a reasonably obvious set of scene environment points (in the form of the cone target). The mean errors obtained are not much higher than the base triangulation error of the camera rigs themselves, which as previously mentioned is up to roughly 2.0 millimetres at the ranges in question.

Experiment 3 provides an example of a case without full convergence; in this situation, the pose estimates within the groups able to converge are still relevant and accurate for those nodes. The convergence time in such cases is much higher because the coarse grouping stage will exhaustively try to bring nodes together, but in the meantime the pose estimates within the calibrated groups are still available.

## 6.5 Virtual Point Set

Since only four physical camera rigs are available, testing scalability to larger networks is impossible in an automatic experiment and difficult to control using the manual methods. Instead, controlled virtual point sets are supplied to the same calibration algorithm implementation to test the scalability metric.

### 6.5.1 Apparatus

Virtual point sets for a given number of nodes are automatically generated by a Python script using the same geometrical libraries as the calibration implementation. First, a specified density of points are placed randomly inside a cylindrical area. The points' positions are then captured within the field of view and coordinate system of "virtual nodes," positioned at random angles and radial distances along the length of the cylinder. Figure 6.9 shows a visualization of the concept with 5 nodes, where the central axis of the cylinder is shown as a line and the virtual node viewpoints are represented as pyramids.

Because these points are free of detection error and occlusions, and because the field of view estimate is inherently perfect, this method has the added advantage of removing other sources of variability from the experiments.

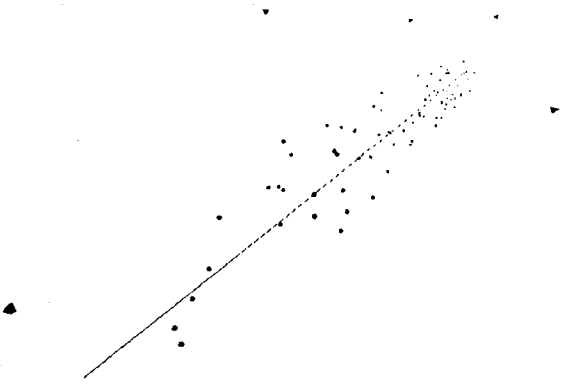

Figure 6.9: Virtual Point Set Generation

### 6.5.2 Procedure

Point sets are generated for 5, 10, 15, 20, and 25 nodes. The total outgoing bandwidth in kilobytes, final size of the matching database in features, and total number of coarse and fine registration executions are recorded.

### 6.5.3 Results

The recorded results of the virtual point set experiments, with node average and node maximum for each measured value, are shown in Table 6.2.

Table 6.2: Virtual Point Set Experiment Results

| Nodes | Total Bandwidth (KB) | Features Stored | Coarse Reg. | Fine Reg. |
|---|---|---|---|---|
| 5 | 26.67 / 29.19 | 29.80 / 37 | 201.60 / 446 | 1.40 / 2 |
| 10 | 64.57 / 79.53 | 93.10 / 111 | 1555.80 / 5461 | 1.70 / 3 |
| 15 | 176.40 / 239.14 | 223.33 / 327 | 14022.13 / 29557 | 1.80 / 3 |
| 20 | 216.08 / 290.48 | 310.10 / 440 | 24250.60 / 57736 | 1.65 / 3 |
| 25 | 280.41 / 417.14 | 369.96 / 581 | 35347.44 / 72757 | 1.92 / 3 |

### Bandwidth Usage

As expected, total bandwidth usage per node increases approximately linearly in relation to the number of nodes in the network (Figure 6.10).

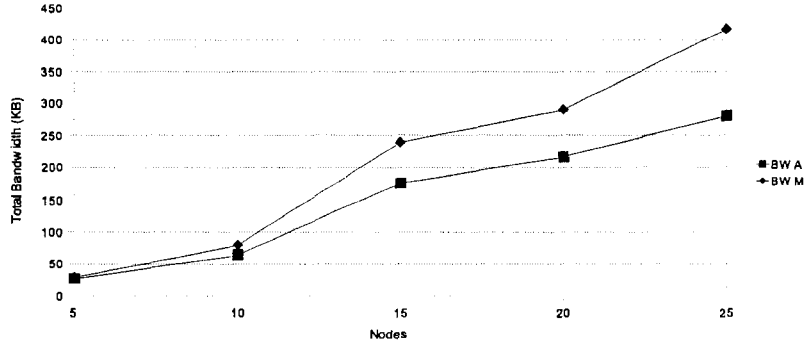


Figure 6.10: Bandwidth Usage in $|\mathbb{N}|$ (Average and Maximum)

This affects different networks in different ways. In a network where the physical medium is shared by all nodes – the worst-case scenario – the total network bandwidth usage is the relevant factor. In that case, the bandwidth usage increases non-linearly: based on these experiments, at roughly $|N|^{2.6}$. However, many networks and topologies are more efficient and therefore able to mitigate this effect. The aforementioned shared-medium case can be considered a ceiling on the increase, while in an ideal case where each pair of nodes has an unshared pairwise link, the per-node bandwidth usage (as in Table 6.2 and Figure 6.10) can be considered a floor.

### Node-Local Storage

The number of features stored at each node increases approximately linearly in relation to the number of nodes (Figure 6.11). Features are very small data (a series of $f$ 3-tuples, an identifier, and a geometric descriptor value), but when scaling to extremely large networks it must be ensured that adequate storage is provided at each node for these features.

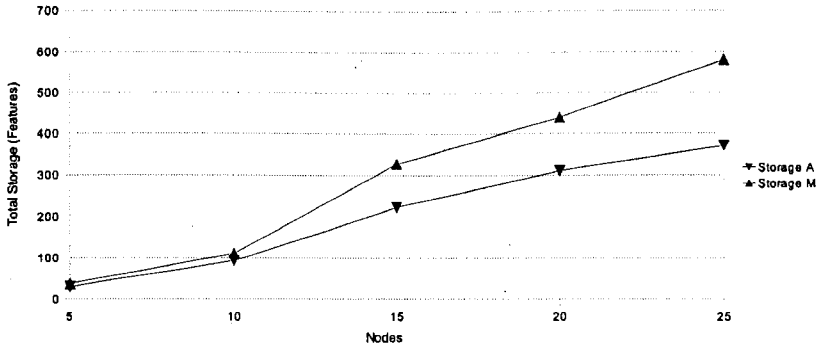


Figure 6.11: Node-Local Storage in $|\mathbb{N}|$ (Average and Maximum)

### Node-Local Processing

The number of coarse registration operations performed at each node increases approximately linearly in relation to the number of nodes (Figure 6.12); as expected, this is proportional to the number of features stored. If processing throughput is the limiting factor, this increase will cause the convergence time to increase linearly with the number of nodes.

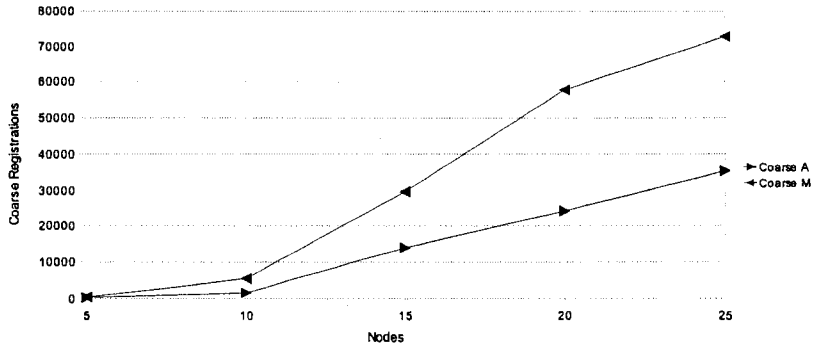


Figure 6.12: Coarse Registration Processing in $|\mathbb{N}|$ (Average and Maximum)

Since this network does not significantly increase the number of nodes whose fields of view overlap as its total number of nodes increases, the number of fine registrations per node does not increase (Figure 6.13). Thus, although fine registration can be considerably more computationally intensive than coarse registration, it does not contribute to increased convergence time.

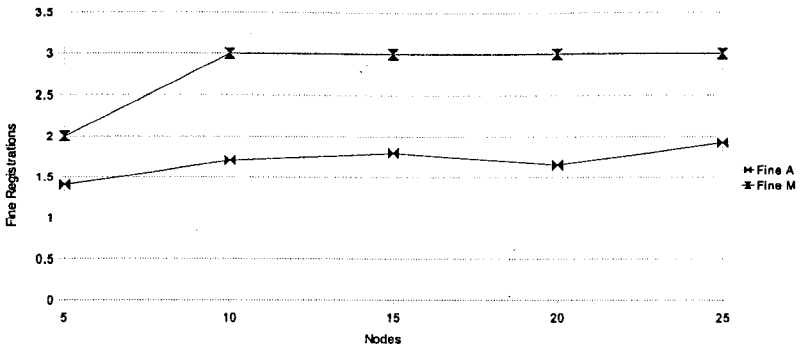


Figure 6.13: Fine Registration Processing in $|\mathbb{N}|$ (Average and Maximum)

### Interpretation

Generally, these results show that the calibration algorithm scales well computationally, with the number of coarse registration operations increasing at no more than $O(n)$ in the number of nodes. Similarly, the storage requirements are $O(n)$ in the number of nodes. Depending on the network specifics, the algorithm may approach linear scalability in bandwidth usage as well, and increases at no more than approximately $O(n^{2.6})$ in the number of nodes in the shared-medium case.

The actual absolute requirements are low enough that, with modern processing, storage, and networking technologies, the algorithm should scale well into the hundreds or thousands of nodes.

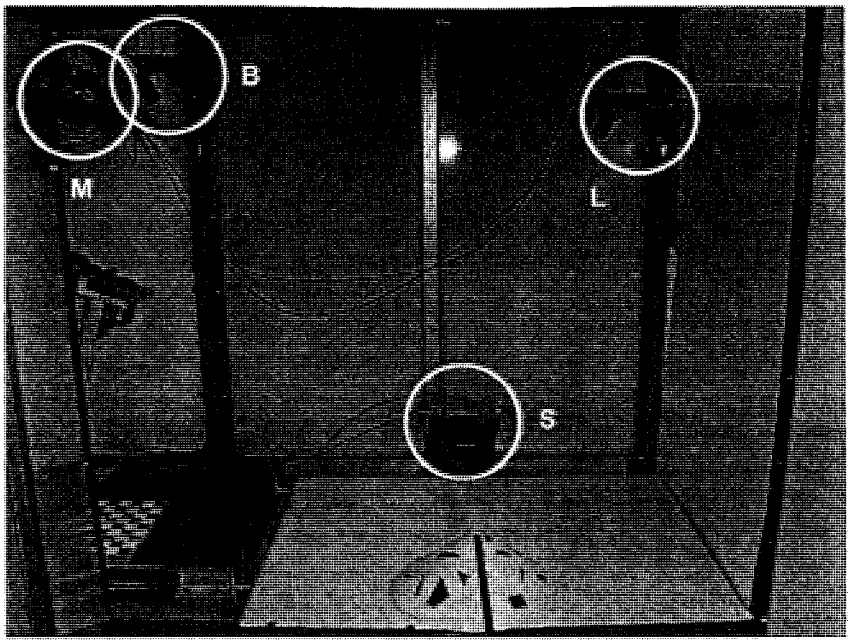


Figure 6.14: Camera Deployment for Automatic Experiment 1

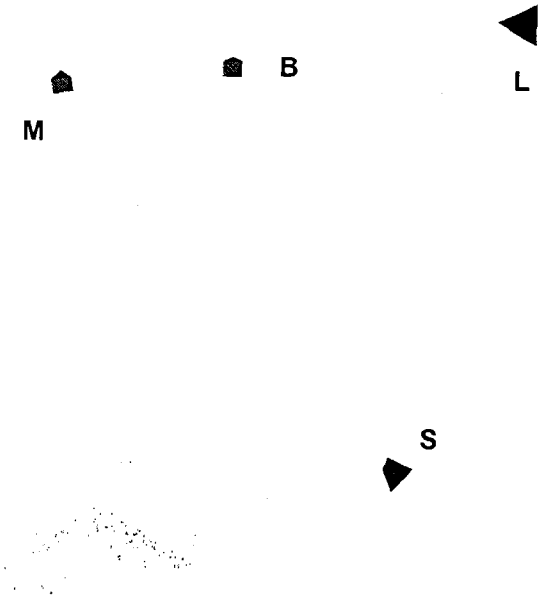


Figure 6.15: Pose Visualization for Automatic Experiment 1

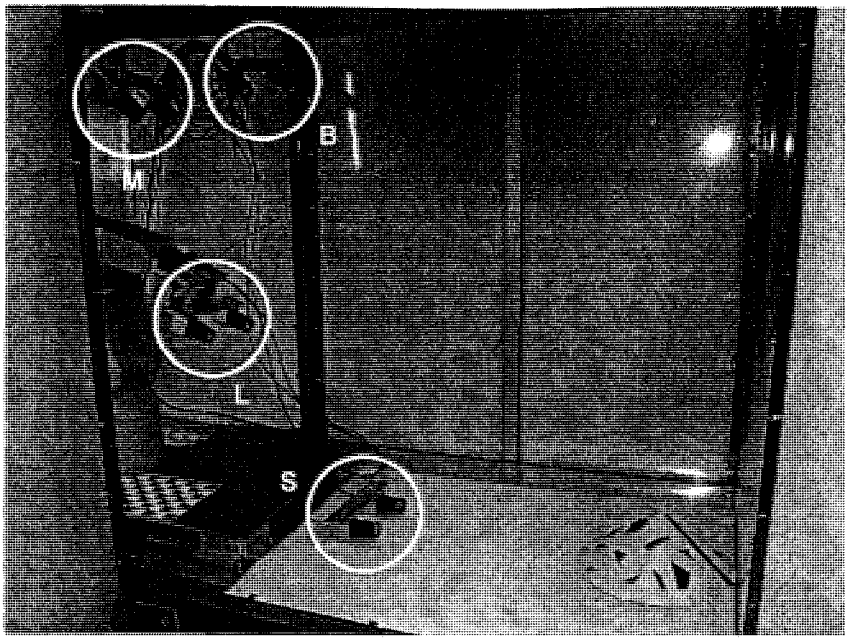


Figure 6.16: Camera Deployment for Automatic Experiment 2

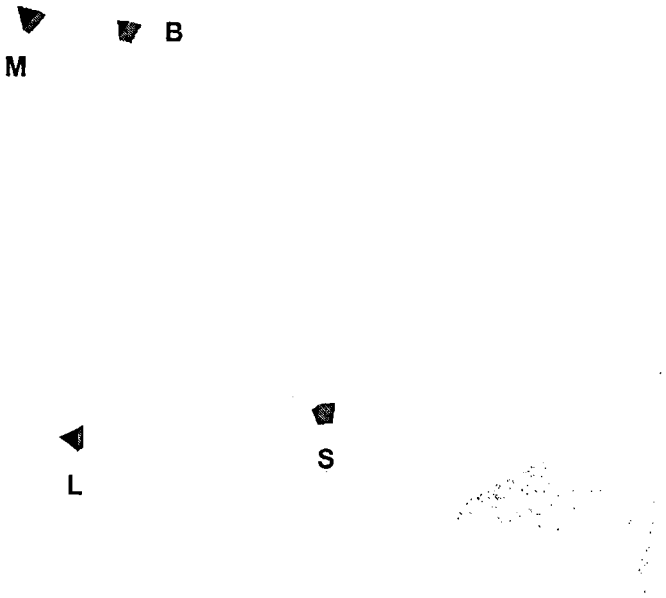


Figure 6.17: Pose Visualization for Automatic Experiment 2

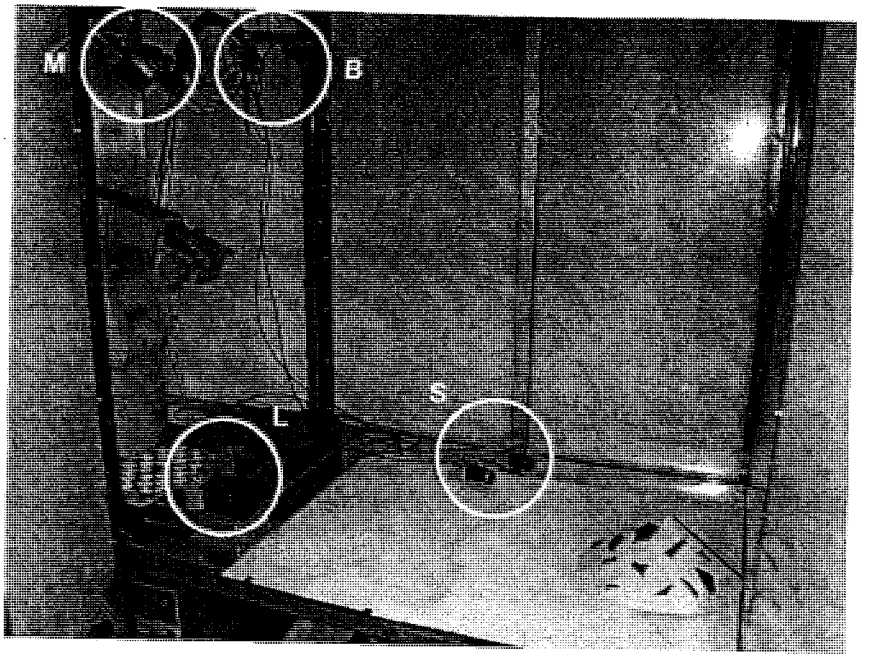


Figure 6.18: Camera Deployment for Automatic Experiment 3

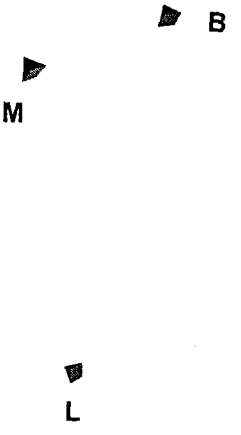


Figure 6.19: Pose Visualization for Automatic Experiment 3

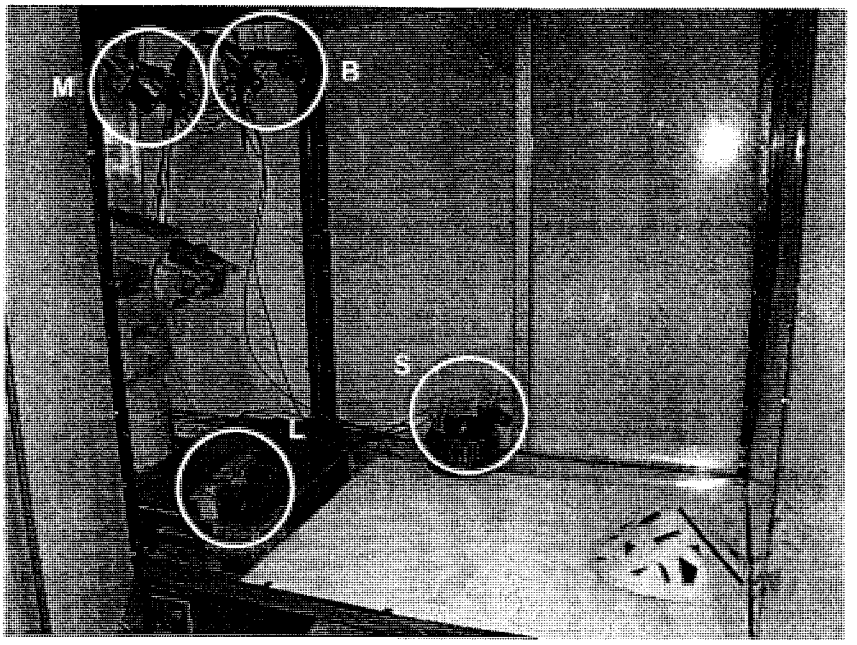


Figure 6.20: Camera Deployment for Automatic Experiment 4

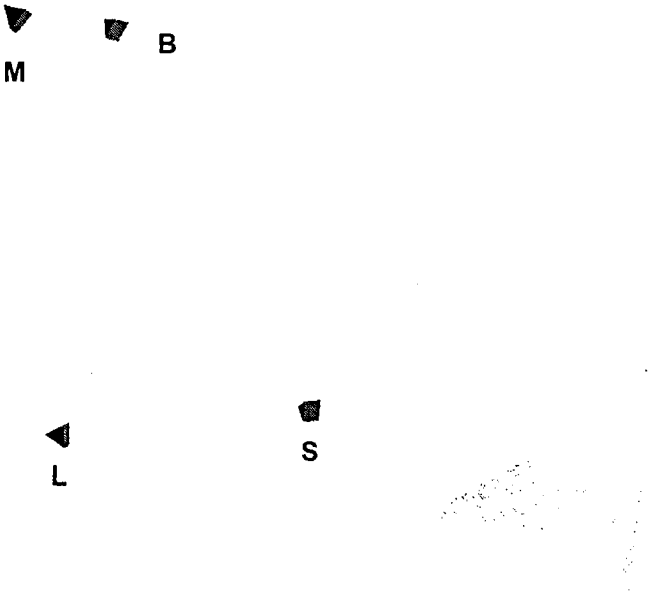


Figure 6.21: Pose Visualization for Automatic Experiment 4

# Chapter 7

# Conclusions

## 7.1 Overview

A feature-based calibration method for distributed smart stereo camera networks has been developed which converges well, provides accurate pairwise orientation, and scales well to large networks. This provides a base upon which to build a full 3D visual sensor network providing primitive data-centric queries, upon which in turn a variety of high-level applications can be developed.

Currently, the algorithm makes it possible for smart stereo camera devices to self-localize and self-orient relative to one another in a distributed fashion, allowing for various subsequent stages of realization for a variety of applications. The immediate opportunity is to provide a generalized framework for building these solutions, which would rest on the underlying assumption that the network is accurately calibrated and can perform 3D reconstruction across multiple views. The preliminary conceptualization is discussed in Section 7.4.5.

The major implementation drawback is the instability of interest point detection in the general case; at present, it is necessary to control the scene somewhat by adding one or more calibration targets for convergence to occur reliably. Improving this situation is an important avenue for future work, and is elaborated upon in Section 7.4.2. Another limitation of the current implementation is its slow performance, often on the order of minutes. This arises from network constraints and the fact that the implementation uses an interpreted language, both of which could be improved on a more specific embedded system, addressed in Section 7.4.1.

## 7.2 Comparison with Existing Work

Since distributed smart stereo cameras are a new concept within a new field, very little exploration has been done as yet in the literature. The major source for this concept to date is the recent work described in [1], and the only other directly comparable calibration algorithm is found in the related work, the Lighthouse method,

in [2]. With this in mind, this calibration algorithm is compared first qualitatively with some of the existing methods for single-camera nodes before moving on to a more quantitative comparison with Lighthouse.

### 7.2.1 Single-Camera Node Methods

The advantages of stereo camera networks are discussed in Chapter 1, but counting this as an implicit advantage, it is still beneficial to consider how this calibration algorithm compares to single-camera methods on other fronts.

While the method in [6] appears to work quite well for networks consisting of single-camera nodes, its major drawback is that it currently requires *a priori* knowledge of the vision graph. The authors state that they plan to automatically generate this via invariant feature matching, but it can be seen from the coarse grouping portion of the calibration algorithm in this work that this is no trivial matter even with robust 3D point sets, let alone with only 2D images. The DALT algorithm in [7] similarly requires knowledge of matching feature points to build microclusters (similar to the vision graph concept), with no explicit robust method for actually doing so. Using robust 3D feature matching and the distributed grouping scheme, this calibration algorithm is able to reliably approximate this information without any *a priori* information.

Methods such as that in [8] rely on the motion of objects through the fields of view of the nodes in order to estimate poses. This limits its applicability to situations where such motion would actually exist, such as in surveillance. Also, the results are not particularly accurate, although they could perhaps be used to initialize one of the previous algorithms [6, 7]. This calibration algorithm makes less assumptions about the contents of the scene, however, and provides highly accurate pairwise pose estimates without requiring external refinement.

The methods of [9] and [10] are fairly robust, but both require the use of markers or beacons in the environment. One of the major strengths of this calibration algorithm is that it does not implicitly depend on any particular scene contents; as long as the scene and the interest point detector combine to provide a good set of points, calibration is possible. The cumbersome or infeasible process of controlling the contents of the scene can be completely avoided in many real cases.

### 7.2.2 Lighthouse

The most similar approach to the one taken in this work, and the only other distributed smart camera network calibration technique using true 3D features published to date, is the Lighthouse method in [2]. This method performs distributed feature matching in a similar way, but its grouping method uses GHTs, which require localization information to function and are ill-suited to feature matching distribution even when methods not requiring localization are used. The method assumes the existence of robust feature detection and geometric hashing; as is seen in this work, this is not a trivial process, and it is necessary that the design of algorithms using these features be coupled with knowledge of methods for feature detection and description.

As a result, it is difficult to compare evaluations of convergence quantitatively. The Lighthouse method appears to assume what would be considered here an unrealistically high degree of feature overlap and an unrealistically low total number of features per node. It is compared to a "flooding" scheme, essentially

equivalent to centralized aggregation and matching of features, and approximates it in terms of convergence. However, the method appears unable to merge the entire network into a single group in any tested case, which is qualitatively poor convergence. By contrast, given more features with less overlap, this calibration algorithm easily converges into a single group.

Accuracy is not evaluated in the Lighthouse method, as the pairwise pose refinement step is considered external. Since similar feature matching methods are used, the accuracy of Lighthouse's results should be comparable to those of the coarse pose estimates in this algorithm, although no data is provided about this either. Consequently, given its internal pairwise pose refinement step, this algorithm is considered to provide superior calibration accuracy.

Evaluated in the context of certain wireless sensor network protocols, notably the geographic routing methods used with GHTs, Lighthouse exhibits good scalability in terms of bandwidth usage. The feature distribution in this method is similar to GHT insertion, and thus would benefit similarly from appropriate routing methods, offering equivalent scalability. Node-local processing and storage requirements in the network sie are not evaluated directly, but again, since the feature matching process is similar, the scalability should be comparable.

## 7.3 Summary of Contributions

The primary contribution of this work is a scalable, general-purpose distributed spatial calibration method and algorithm for smart stereo camera networks. This is the most complex and important building block of a true 3D sensing network.

In conjunction with the development of this method, algorithms for interest point detection and registration have been surveyed, providing a truly practical implementation and also exposing areas of improvement in these problems.

Another important contribution is an examination of the significance of and relationships between the communication graph, vision graph, and calibration graph. Visual sensor networks differ fundamentally from other networks, and these graphs provide a theoretical basis for modelling them.

## 7.4 Future Work

### 7.4.1 Embedded Implementation

Currently, the practical applications of this work are limited without a concrete underlying platform. Now that the fundamental requirements are established, a next step towards practicality is the design and implementation of a physical embedded smart stereo camera device capable of providing the underlying networking system in an ad-hoc manner and of executing distributed collaborative algorithms such as this calibration method.

### 7.4.2 Improved Feature Reliability

The challenge in using 3D feature-based methods for calibration and other components of a distributed smart stereo camera network's sensing system is the inherent difficulty of obtaining reliable feature data from 2D images of the scene. Here, standard image-based detection and correspondence methods have been used, which is cited as the most prominent drawback in the calibration algorithm. There are, fortunately, many possibilities for improving this situation.

The analysis of existing image-based interest point detection algorithms in [25] has shown that none are fully adequate for 3D transformations, with the best of them invariant only to affine changes. However, these limitations are primarily imposed by the fact that the algorithms are provided only a single image as input. With stereo cameras, new algorithms could avail themselves of the much richer information and provide far more robust interest point detection relevant to a 3D context. Some work has been done attempting to use stereo information to enhance the relevancy and distribution of interest points in [26, 27], using *epipolar gradients* – information not available in single images – to enhance basic Harris-based corner detection. More directly, some investigation into developing a 3D rigid interest point detector from stereo images has been presented in [63].

With the availability of interest point descriptors arising directly from the detector, the feature matching portion of calibration could be greatly improved as well, taking advantage of more than just the coordinates of the interest points in categorizing and matching features.

### 7.4.3 Tiered Calibration for Large Networks

At certain large network sizes, the increased convergence time or storage requirements of the algorithm become infeasible. It may thus be desirable to divide the network into a series of subgroups which are known to be at least somewhat contiguous in terms of the portion of the total network coverage they represent, perform calibration within these subgroups, and then perform a second calibration for the full network with the calibrated subgroups initialized as groups (using their fine pose estimates to initialize the coarse pose estimates). This might be cascaded an arbitrary number of times, thus providing essentially unlimited scalability.

Such an adaptation would be possible using the existing calibration algorithm essentially unmodified, requiring only a simple higher-level mechanism to initialize and coordinate the successive calibrations, and an assumption that the deployment locations and orientations of the nodes are known to some degree.

### 7.4.4 Dynamic Calibration

It is desirable for a calibration algorithm for distributed smart stereo camera networks to adapt to changes in node presence and pose. Such adaptation should be automatic, both for small changes in node pose (e.g. panning or drift) and for large changes in the network (e.g. adding, removing, or relocating nodes).

Dynamic calibration is currently achievable only in a "manual" way; if, after a network is calibrated, nodes are added to or removed from the network, or change position or orientation, the calibration algorithm can be reinitialized with all the group structure and pose estimates which are still valid from the previous cal-

ibration. However, no mechanism has been discussed for how to detect such changes or how to automatically reinitialize the network. Such a mechanism is a candidate for future work directly applicable to this method.

True dynamic calibration, where the nodes constantly adjust their relative pose estimates based on new information in real time, is the ultimate goal. It is conceivable that a second calibration algorithm could take over upkeep for relatively small pose adjustments after the primary calibration is complete, so the aforementioned reinitialization would be necessary only for large adjustments or changes in the network composition.

### 7.4.5 Basis for a 3D Sensing Network

In order to realize the ultimate goal of providing a framework upon which various distributed smart stereo camera network applications can be built, the basic services must be expanded beyond spatial calibration alone. Providing temporal synchronization and a basic space-time query system, as proposed in [1], is the greater context of this work.

# Appendix A

# Glossary of Terms

**coarse registration**
See *registration.*

**DARCES**
Data-Aligned Rigidity-Constrained Exhaustive Search, a method for fully-contained coarse *registration* [13] used in feature matching. Traditionally used with *RANSAC* for registration of partially overlapping data sets.

**feature**
A subset of small fixed size selected from the *point set* for feature matching.

**fine registration**
See *registration.*

**group**
A group of nodes agreeing on a common leader node within the group. See also *group leader*, *group coarse pose.*

**group coarse pose**
The coarse pose estimate of a node relative to the leader of its current group. See also *group leader.*

**group leader**
A node which provides a common coordinate reference for *group coarse pose* estimates. Also, the node within a group responsible for performing *group merge* operations.

**group merge**
A transitive operation involving the *composition* of coarse *pose* estimates which brings two groups together

into a single group.

**ICP**
Iterative Closest Point, a method for fine registration [14]. See also *TrICP*.

**interest point detection**
The process of detecting salient features, such as corners, in 2D images. A variety of methods exist.

**leader**
See *group leader*.

**match**
Two *features* are said to match when they are geometrically similar to within some threshold. May also refer to the resultant transformation (pose) returned by coarse registration upon detecting a match. Defined in Section 3.2.3.

**merge**
See *group merge*.

**node**
A smart stereo camera device in the network.

**point set**
The set of all 3D points locally detected (via *interest point detection*) and triangulated at a *node*.

**pose**
A rigid Euclidean transformation describing an object's location and orientation.

**RANSAC**
Random Sample Consensus, an iterative method to estimate parameters of a mathematical model from a set of observed data which contains outliers.

**registration**
The process of transforming two or more visually acquired data sets into a common coordinate system. A variety of methods exist, generally divided into coarse registration and fine registration algorithms.

**relative pose**
The rigid Euclidean transformation from the local coordinate system of one node to that of another node. See also *pose*.

**repeatability**
A metric by which *interest point detection* algorithms are evaluated, which describes their stability in detecting points under varying conditions and from various viewpoints.

**rotation matrix**
A $n \times n$ real orthogonal matrix corresponding to a geometric rotation about a fixed origin in $n$-dimensional Euclidean space.

**similarity condition**
A condition specifying that two *features* must be sufficiently similar in order to attempt coarse registration for the purpose of feature matching. Defined in Section 4.3.3.

**translation vector**
A $n$-element vector corresponding to a geometric translation in $n$-dimensional Euclidean space.

**TrICP**
Trimmed Iterative Closest Point, a method for fine *registration* of partially overlapping data sets [15] based on *ICP*.

# Appendix B

# Software Source Code

## B.1 Distributed Calibration (Python)

The distributed calibration software is written in the Python programming language [65], and uses the NumPy numeric library [66] for some of the linear algebra computations. It is split into three parts: node.py, which runs the multi-threaded core distributed calibration algorithm; geometry.py, which provides classes for geometrical mappings; and registration.py, which performs the coarse and fine registration algorithms.

### B.1.1 Node Program

**node.py**

```
"""Distributed Smart Stereo Network Calibration Algorithm"""
__author__ = 'Aaron Mavrinac'
__version__ = '1.0'

import time
import sys
import csv
import math
import numpy
import pickle
import random
import socket
import threading
import geometry
import registration
```

```
##############################################################################
# CONSTANTS
##############################################################################

NODE_CONE_ANGLE = ( 1.0 / 3.0 ) * math.pi
NODE_CONE_LENGTH = 3000.0
NODE_FEATURE_SIZE = 4
NODE_FEATURE_DELAY = 0.08
NODE_DIFF_THRESH = 10.0
NODE_MERGE_THRESH = 3
NODE_CONSIST_THRESH = 5.0
COARSE_THRESH = 2.8
FINE_TE = 1.0
FINE_TR = 0.01
FINE_LAMBDA = 2.0
FINE_EMAX = 100.0


##############################################################################
# GENERAL FUNCTIONS
##############################################################################

# timestamp printing function
timestamp = lambda : time.strftime( "[%H:%M:%S]" )

# factorial function
fac = lambda n : [ 1, 0 ][ n > 0 ] or fac( n - 1 ) * n

# dictionary length function
dictlen = lambda x : sum( map( lambda k : len( x[ k ] ), x.keys() ) )

# unbiased deterministic node selector
def nodeselect( a, b ):
  mod = ( net.idsubscript( b ) - net.idsubscript( a ) ) % 2
  if ( mod and b > a ) or ( not mod and b < a ):
    return True
  else:
    return False
```

```
# unique combination generator
def uniquecombinations( items, n ):
  if n == 0:
    yield []
  else:
    for i in xrange( len( items ) ):
      for cc in uniquecombinations( items[ i + 1: ], n - 1 ):
        yield [ items[ i ] ] + cc

# network class
class network( list ):
  def __init__( self ):
    self.outaccount = {}
    list.__init__( self )
  def idsubscript( self, node ):
    """Returns the subscript for a node given its node ID"""
    for i in range( len( self ) ):
      if self[ i ][ 0 ] == node:
        return i
  def send( self, node, msg ):
    """Sends a pickled message to another node"""
    outmsg = pickle.dumps( msg )
    client = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
    try:
      client.connect( self.hostport( node ) )
      client.send( outmsg )
      client.close()
      # bandwidth accounting
      t = int( time.time() )
      if self.outaccount.has_key( t ):
        self.outaccount[ t ] += len( outmsg )
      else:
        self.outaccount[ t ] = len( outmsg )
    except:
      print timestamp(), "Failed to send", msg[ 0 ], "message to [", node, "]."
  def hostport( self, node ):
    """Returns the hostname and port number for a node"""
    i = self.idsubscript( node )
```

```
    return ( self[ i ][ 1 ], self[ i ][ 2 ] )


################################################################################
# COARSE PROCESS 1: Feature Selection
################################################################################

class coarse_feature_thread( threading.Thread ):
  def run( self ):
    # generate the feature list
    rfeatures = []
    for c in uniquecombinations( range( len( globals()['points'] ) ), \
    globals()['NODE_FEATURE_SIZE'] ):
      cfeat = []
      for r in c:
        cfeat.append( globals()['points'][ r ] )
      rfeatures.append( [ c, self.geometric_hash( cfeat ) ] )
    rfeatures.sort( lambda x, y : cmp( x[ 1 ], y[ 1 ] ) )
    # pre-bin all features
    features = {}
    for node in globals()['net']:
      features[ node[ 0 ] ] = []
      for rf in rfeatures:
        if rf[ 1 ] > node[ 4 ]:
          break
        if rf[ 1 ] > node[ 3 ]:
          features[ node[ 0 ] ].append( rf )
    del rfeatures
    # wait for other nodes to come online
    print timestamp(), "Waiting for other nodes..."
    waitlist = network()
    for node in globals()['net']:
      if node[ 0 ] != globals()['nodeid']:
        waitlist.append( node )
    while( len( waitlist ) ):
      for node in waitlist:
        ping = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
        try:
          ping.connect( waitlist.hostport( node[ 0 ] ) )
```

```
        ping.send( pickle.dumps( [ 'ping' ] ) )
        ping.close()
        del waitlist[ waitlist.idsubscript( node[ 0 ] ) ]
      except:
        time.sleep( 1.0 )
  # periodically send out features
  print timestamp(), "Starting coarse feature dissemination."
  i = 0
  while len( globals()['group'] ) < len( globals()['net'] ) \
  and dictlen( features ) > 0:
    for node in features.keys():
      if len( features[ node ] ) > 0:
        feature = [ 'cfeature', globals()['nodeid'], i ]
        rf = features[ node ].pop( 0 )
        feature.append( rf[ 1 ] )
        for r in rf[ 0 ]:
          feature.append( globals()['points'][ r ] )
        globals()['net'].send( node, feature )
        i += 1
        time.sleep( globals()['NODE_FEATURE_DELAY'] + 0.0001 * i )
      else:
        del features[ node ]
  feature = [ 'cfeature' ]
  for node in globals()['net']:
    globals()['net'].send( node[ 0 ], feature )
  print timestamp(), "Stopping coarse feature dissemination process."
  globals()['net'].send( globals()['nodeid'], [ 'die' ] )
def geometric_hash( self, feature ):
  """Simple geometric descriptor (total Euclidean distance to centroid)"""
  centroid = geometry.point( 0, 0, 0 )
  gdesc = 0.0
  for p in feature:
    centroid += p
  centroid /= float( len( feature ) )
  for p in feature:
    gdesc += abs( centroid.euclidean( p ) )
  return gdesc
```

```
##############################################################################
# COARSE PROCESS 2: Feature Matching
##############################################################################
class coarse_matching_thread( threading.Thread ):
  def run( self ):
    dc = 0
    matchdb = []
    while len( globals()['group'] ) < len( globals()['net'] ) \
    and dc < len( globals()['net'] ):
      # grab next message in the queue
      globals()['cvqueue'][ 'cfeature' ].acquire()
      while len( globals()['msgqueue'][ 'cfeature' ] ) == 0:
        globals()['cvqueue'][ 'cfeature' ].wait()
      feature = globals()['msgqueue'][ 'cfeature' ].pop( 0 )
      globals()['cvqueue'][ 'cfeature' ].release()
      if len( feature ) == 1:
        dc += 1
        continue
      # process feature matching
      for fext in matchdb:
        if abs( feature[ 3 ] - fext[ 2 ] ) <= globals()['NODE_DIFF_THRESH'] \
        and not feature[ 1 ] == fext[ 0 ] \
        and not ( feature[ 1 ] in globals()['group'] \
        and fext[ 0 ] in globals()['group'] ):
          rpose = registration.coarse_registration( fext[ 3: ], \
          feature[ 4: ], globals()['COARSE_THRESH'] )
  globals()['acc_coarse'] += 1
          if rpose.nonzero():
            print timestamp(), "Matched features between node [", \
            feature[ 1 ], "] and node [", fext[ 0 ], "]."
            if nodeselect( fext[ 0 ], feature[ 1 ] ):
              match = [ 'cmatch', feature[ 1 ], feature[ 2 ] + fext[ 1 ], \
              rpose ]
              globals()['net'].send( fext[ 0 ], match )
            else:
              match = [ 'cmatch', fext[ 0 ], fext[ 1 ] + feature[ 2 ], -rpose ]
              globals()['net'].send( feature[ 1 ], match )
      matchdb.append( feature[ 1: ] )
    match = [ 'cmatch' ]
```

```
    for node in globals()['net']:
      globals()['net'].send( node[ 0 ], match )
    print timestamp(), "Stopping coarse matching process (stored", \
    len( matchdb ), "features)."
    globals()['acc_matchdb'] = len( matchdb )
    globals()['net'].send( globals()['nodeid'], [ 'die' ] )


################################################################################
# COARSE PROCESS 3: Match Processing
################################################################################

class coarse_matchproc_thread( threading.Thread ):
  def run( self ):
    dc = 0
    matches = {}
    for node in globals()['net']:
      matches[ node[ 0 ] ] = []
    while dc < len( globals()['net'] ):
      # grab next message in the queue
      globals()['cvqueue'][ 'cmatch' ].acquire()
      while len( globals()['msgqueue'][ 'cmatch' ] ) == 0:
        globals()['cvqueue'][ 'cmatch' ].wait()
      match = globals()['msgqueue'][ 'cmatch' ].pop( 0 )
      globals()['cvqueue'][ 'cmatch' ].release()
      if len( match ) == 1:
        dc += 1
        continue
      # store and process feature match
      if not matches[ match[ 1 ] ] == 'done' \
      and not match[ 1 ] in globals()['group'] \
      and not match[ 2 ] in map( lambda x : x[ 0 ], matches[ match[ 1 ] ] ):
        print timestamp(), "Received a match with node [", match[ 1 ], "]."
        if len( matches[ match[ 1 ] ] ) >= \
        ( globals()['NODE_MERGE_THRESH'] - 1 ):
          # try to find enough consistent matches to merge
          for matchsubset in uniquecombinations( matches[ match[ 1 ] ], \
          globals()['NODE_MERGE_THRESH'] - 1 ):
            matchsubset.append( [ match[ 2 ], match[ 3 ] ] )
```

```
          # calculate the average pose
          avgpose = geometry.pose( 0, 0 )
          avgtheta = 0.0
          avgphi = 0.0
          avgpsi = 0.0
          for rpose in map( lambda x : x[ 1 ], matchsubset ):
            for i in range( 3 ):
              avgpose.T[ i ] += rpose.T[ i ] / \
              float( globals()['NODE_MERGE_THRESH'] )
            avgtheta += rpose.om()[ 0 ] / \
            float( globals()['NODE_MERGE_THRESH'] )
            avgphi += rpose.om()[ 1 ] / \
            float( globals()['NODE_MERGE_THRESH'] )
            avgpsi += rpose.om()[ 2 ] / \
            float( globals()['NODE_MERGE_THRESH'] )
          avgpose.generate( avgpose.T.x, avgpose.T.y, avgpose.T.z, \
          avgtheta, avgphi, avgpsi )
          # check if matches are consistent
          flag = False
          for rpose in map( lambda x : x[ 1 ], matchsubset ):
            if rpose.map( globals()['pointc'] ).euclidean( avgpose.map( \
            globals()['pointc'] ) ) > globals()['NODE_CONSIST_THRESH']:
              flag = True
              break
          if flag:
            continue
          matches[ match[ 1 ] ] = 'done'
          # send the merge message to the other node
          print timestamp(), "Forwarding pose from", \
          globals()['NODE_MERGE_THRESH'], "matches to leader [", \
          globals()['groupid'], "]."
          pose = [ 'cpose', globals()['nodeid'], match[ 1 ], \
          globals()['coarsepose'][ globals()['groupid'] ], avgpose ]
          globals()['net'].send( globals()['groupid'], pose )
          break
      # add the match to the set
      if not matches[ match[ 1 ] ] == 'done':
        matches[ match[ 1 ] ].append( [ match[ 2 ], match[ 3 ] ] )
  pose = [ 'cpose' ]
```

```
    for node in globals()['net']:
      globals()['net'].send( node[ 0 ], pose )
    print timestamp(), "Stopping coarse match processing process."
    globals()['net'].send( globals()['nodeid'], [ 'die' ] )


##############################################################################
# COARSE PROCESS 4: Group Merge Initiator
##############################################################################

class coarse_pose_thread( threading.Thread ):
  def run( self ):
    dc = 0
    last = globals()['nodeid']
    while dc < len( globals()['net'] ) \
    and globals()['nodeid'] == globals()['groupid']:
      # grab next message in the queue
      globals()['cvqueue'][ 'cpose' ].acquire()
      while len( globals()['msgqueue'][ 'cpose' ] ) == 0:
        globals()['cvqueue'][ 'cpose' ].wait()
      pose = globals()['msgqueue'][ 'cpose' ].pop( 0 )
      globals()['cvqueue'][ 'cpose' ].release()
      if len( pose ) == 1:
        dc += 1
        continue
      if len( globals()['group'] ) == len( globals()['net'] ):
        break
      if not pose[ 2 ] in globals()['group']:
        globals()['mergelock'].acquire()
        if pose[ 2 ] != last:
          # if not group leader, forward to group leader
          if globals()['nodeid'] != globals()['groupid']:
            print timestamp(), "Re-forwarding pose from node [", pose[ 1 ], \
            "] to leader node [", globals()['groupid'], "]."
            globals()['net'].send( globals()['groupid'], pose )
            globals()['mergelock'].release()
            break
          # send a merge message to the other group
          print timestamp(), "Initiating merge into group containing node [", \
```

```
  pose[ 2 ], "]."
  merge = [ 'cmerge', globals()['nodeid'], globals()['group'], \
  geometry.pose( 0, 0 ) ]
  globals()['net'].send( pose[ 2 ], merge )
  last = pose[ 2 ]
# wait for acknowledgement from the other group's leader
globals()['cvqueue'][ 'cack' ].acquire()
while len( globals()['msgqueue'][ 'cack' ] ) == 0:
  globals()['cvqueue'][ 'cack' ].wait()
ack = globals()['msgqueue'][ 'cack' ].pop( 0 )
globals()['cvqueue'][ 'cack' ].release()
# check for merge thread preempt
if len( ack ) == 1:
  print timestamp(), "Merge initiation preempted, deferring."
  # reinsert the pose message
  globals()['cvqueue'][ 'cpose' ].acquire()
  globals()['msgqueue'][ 'cpose' ].insert( 0, pose )
  globals()['cvqueue'][ 'cpose' ].release()
  globals()['mergelock'].release()
  # give the merge thread a chance to acquire the lock
  time.sleep( 1.0 )
  continue
# update former own group
print timestamp(), "Merge with group [", ack[ 1 ], \
"] acknowledged, merging."
update = [ 'cupdate', globals()['nodeid'], ack[ 1 ], ack[ 2 ], \
ack[ 3 ], pose[ 4 ], pose[ 3 ] ]
for node in globals()['group']:
  if node != globals()['nodeid']:
    globals()['net'].send( node, update )
# update self
globals()['groupid'] = ack[ 1 ]
for node in ack[ 2 ]:
  globals()['group'].append( node )
coarsepose[ ack[ 1 ] ] = ( ( -pose[ 3 ] ) + pose[ 4 ] ) + ack[ 3 ]
globals()['mergelock'].release()
globals()['groupupdate'].set()
print timestamp(), "Group [", globals()['groupid'], "] now contains", \
globals()['group']
```

```
    # forward all future poses to leader
    while dc < len( globals()['net'] ):
      # grab next message in the queue
      globals()['cvqueue'][ 'cpose' ].acquire()
      while len( globals()['msgqueue'][ 'cpose' ] ) == 0:
        globals()['cvqueue'][ 'cpose' ].wait()
      pose = globals()['msgqueue'][ 'cpose' ].pop( 0 )
      globals()['cvqueue'][ 'cpose' ].release()
      if len( pose ) == 1:
        dc += 1
        continue
      globals()['net'].send( globals()['groupid'], pose )
    merge = [ 'cmerge' ]
    for node in globals()['net']:
      globals()['net'].send( node[ 0 ], merge )
    print timestamp(), "Stopping coarse group merge initiator process."
    globals()['net'].send( globals()['nodeid'], [ 'die' ] )


################################################################################
# COARSE PROCESS 5: Group Merge Responder
################################################################################

class coarse_merge_thread( threading.Thread ):
  def run( self ):
    dc = 0
    while dc < len( globals()['net'] ) \
    and globals()['nodeid'] == globals()['groupid']:
      # grab next message in the queue
      globals()['cvqueue'][ 'cmerge' ].acquire()
      while len( globals()['msgqueue'][ 'cmerge' ] ) == 0:
        globals()['cvqueue'][ 'cmerge' ].wait()
      merge = globals()['msgqueue'][ 'cmerge' ].pop( 0 )
      globals()['cvqueue'][ 'cmerge' ].release()
      if len( merge ) == 1:
        dc += 1
        continue
      # preempt after a random period to avoid deadlocks
      if not globals()['mergelock'].acquire( False ):
```

```
      tmr = threading.Timer( random.randrange( 3, 30, 3 ), self.preempt )
      tmr.start()
      globals()['mergelock'].acquire()
      tmr.cancel()
    # make sure the offer is still valid
    if merge[ 1 ] in globals()['group']:
      continue
    # if not group leader, forward to group leader
    if globals()['nodeid'] != globals()['groupid']:
      print timestamp(), "Re-forwarding merge from node [", merge[ 1 ], \
      "] to leader node [", globals()['groupid'], "]."
      merge[ 3 ] += globals()['coarsepose'][ globals()['groupid'] ]
      globals()['net'].send( globals()['groupid'], merge )
      globals()['mergelock'].release()
      break
    # send acknowledge
    ack = [ 'cack', globals()['nodeid'], globals()['group'] ]
    ack.append( merge[ 3 ] )
    globals()['net'].send( merge[ 1 ], ack )
    # update own group
    print timestamp(), "Updating group [", globals()['groupid'], \
    "] memberships with group [", merge[ 1 ], "]."
    update = [ 'cupdate', globals()['groupid'], globals()['groupid'], \
    merge[ 2 ] ]
    for i in range( 3 ):
      update.append( geometry.pose( 0, 0 ) )
    for node in globals()['group']:
      if node != globals()['nodeid']:
        globals()['net'].send( node, update )
    # update self
    for node in merge[ 2 ]:
      globals()['group'].append( node )
    globals()['mergelock'].release()
    globals()['groupupdate'].set()
    print timestamp(), "Group [", globals()['groupid'], "] now contains", \
    globals()['group']
  # forward all future merges to leader
  while dc < len( globals()['net'] ):
    # grab next message in the queue
```

```
      globals()['cvqueue'][ 'cmerge' ].acquire()
      while len( globals()['msgqueue'][ 'cmerge' ] ) == 0:
        globals()['cvqueue'][ 'cmerge' ].wait()
      merge = globals()['msgqueue'][ 'cmerge' ].pop( 0 )
      globals()['cvqueue'][ 'cmerge' ].release()
      if len( merge ) == 1:
        dc += 1
        continue
      merge[ 3 ] += globals()['coarsepose'][ globals()['groupid'] ]
      globals()['net'].send( globals()['groupid'], merge )
    print timestamp(), "Stopping coarse group merge responder process."
    globals()['net'].send( globals()['nodeid'], [ 'die' ] )
  def preempt( self ):
    """Sends a preemption message to this node's initiator process."""
    print timestamp(), "Preempting to process merge."
    ack = [ 'cack' ]
    globals()['net'].send( globals()['nodeid'], ack )


################################################################################
# COARSE PROCESS 6: Group Update
################################################################################

class coarse_update_thread( threading.Thread ):
  def run( self ):
    while globals()['nodeid'] == globals()['groupid'] \
    and len( globals()['group'] ) < len( globals()['net'] ):
      globals()['groupupdate'].wait()
    while len( globals()['group'] ) < len( globals()['net'] ):
      # grab next message in the queue
      globals()['cvqueue'][ 'cupdate' ].acquire()
      t = False
      while not t:
        for i in range( len( globals()['msgqueue'][ 'cupdate' ] ) ):
          if globals()['msgqueue'][ 'cupdate' ][ i ][ 1 ] == \
          globals()['groupid']:
            update = globals()['msgqueue'][ 'cupdate' ].pop( i )
            t = True
            break
```

```
        if not t:
          globals()['cvqueue'][ 'cupdate' ].wait()
      globals()['cvqueue'][ 'cupdate' ].release()
      if len( update ) == 2:
        break
      # process group update
      print timestamp(), "Got group update for group [", update[ 2 ], "]."
      globals()['coarsepose'][ update[ 2 ] ] = \
      ( ( globals()['coarsepose'][ globals()['groupid'] ] - update[ 6 ] ) + \
      update[ 5 ] ) + update[ 4 ]
      globals()['groupid'] = update[ 2 ]
      for node in update[ 3 ]:
        globals()['group'].append( node )
      globals()['groupupdate'].set()
      print timestamp(), "Group [", globals()['groupid'], "] now contains", \
      globals()['group']
    print timestamp(), "Stopping coarse group update process."
    globals()['net'].send( globals()['nodeid'], [ 'die' ] )


################################################################################
# FINE PROCESS 1: Pose Refinement Initiator
################################################################################

class fine_init_thread( threading.Thread ):
  def run( self ):
    offset = 0
    while offset < len( globals()['net'] ) and ( globals()['threadcount'] > 4 \
    or offset < len( globals()['group'] ) ):
      globals()['groupupdate'].wait()
      globals()['groupupdate'].clear()
      glen = len( globals()['group'] )
      for node in globals()['group'][ offset:glen ]:
        if nodeselect( globals()['nodeid'], node ) \
        and not globals()['finepose'].has_key( node ):
          print timestamp(), "Initiating fine calibration with node [", node, \
          "]."
          init = [ 'finit', globals()['nodeid'], globals()['groupid'], \
          globals()['coarsepose'][ globals()['groupid'] ] ]
```

```
        globals()['net'].send( node, init )
      offset = glen
    init = [ 'finit' ]
    for node in globals()['group']:
      globals()['net'].send( node, init )
    print timestamp(), "Stopping pose refinement initiator process."
    globals()['net'].send( globals()['nodeid'], [ 'die' ] )


################################################################################
# FINE PROCESS 2: Pose Refinement Responder
################################################################################

class fine_respond_thread( threading.Thread ):
  def run( self ):
    dc = 0
    while dc < len( globals()['net'] ) \
    or len( globals()['msgqueue'][ 'finit' ] ) > 0:
      # grab next message in the queue
      globals()['cvqueue'][ 'finit' ].acquire()
      while len( globals()['msgqueue'][ 'finit' ] ) == 0:
        globals()['cvqueue'][ 'finit' ].wait()
      init = globals()['msgqueue'][ 'finit' ].pop( 0 )
      globals()['cvqueue'][ 'finit' ].release()
      if len( init ) == 1:
        dc += 1
        continue
      # process fine init
      while not globals()['coarsepose'].has_key( init[ 2 ] ):
        globals()['groupupdate'].wait()
      relpose = globals()['coarsepose'][ init[ 2 ] ] - init[ 3 ]
      fpoints = []
      for p in globals()['points']:
        p = relpose.map( p )
        if p.z < globals()['NODE_CONE_LENGTH'] \
        and p.euclidean( geometry.point( 0, 0, p.z ) ) < \
        ( p.z * math.tan( globals()['NODE_CONE_ANGLE'] ) ):
          fpoints.append( p )
      if len( fpoints ) > 3:
```

```
        print timestamp(), "Responding to fine calibration with node [", \
        init[ 1 ], "],", len( fpoints ), "points."
        respond = [ 'frespond', globals()['nodeid'], init[ 2 ], \
        globals()['coarsepose'][ init[ 2 ] ], fpoints ]
        globals()['net'].send( init[ 1 ], respond )
      else:
        print timestamp(), "Not enough points shared with node [", init[ 1 ], \
        "]."
    respond = [ 'frespond' ]
    for node in globals()['group']:
      globals()['net'].send( node, respond )
    print timestamp(), "Stopping pose refinement responder process."
    globals()['net'].send( globals()['nodeid'], [ 'die' ] )


################################################################################
# FINE PROCESS 3: Pose Refinement Registration
################################################################################

class fine_registration_thread( threading.Thread ):
  def run( self ):
    dc = 0
    while dc < len( globals()['net'] ) \
    or len( globals()['msgqueue'][ 'frespond' ] ) > 0:
      # grab next message in the queue
      globals()['cvqueue'][ 'frespond' ].acquire()
      while len( globals()['msgqueue'][ 'frespond' ] ) == 0:
        globals()['cvqueue'][ 'frespond' ].wait()
      respond = globals()['msgqueue'][ 'frespond' ].pop( 0 )
      globals()['cvqueue'][ 'frespond' ].release()
      if len( respond ) == 1:
        dc += 1
        continue
      # process fine response
      relpose = globals()['coarsepose'][ respond[ 2 ] ] - respond[ 3 ]
      fpoints = []
      for p in globals()['points']:
        q = relpose.map( p )
        if q.z < globals()['NODE_CONE_LENGTH'] \
```

```
        and q.euclidean( geometry.point( 0, 0, q.z ) ) < \
        ( q.z * math.tan( globals()['NODE_CONE_ANGLE'] ) ):
          fpoints.append( p )
      if len( fpoints ) > 3:
        print timestamp(), "Computing pairwise fine calibration with node [", \
        respond[ 1 ], "]."
        emin = float( 10 ** 18 )
        for zeta in range( 40, 101 ):
          zeta /= 100.0
          fpose, e = registration.fine_registration( respond[ 4 ], fpoints, \
          zeta, globals()['FINE_TE'], globals()['FINE_TR'] )
globals()['acc_fine'] += 1
          e *= ( zeta ** -( 1.0 + globals()['FINE_LAMBDA'] ) )
          if e < emin:
            emin = e
            zetamin = zeta
            globals()['finepose'][ respond[ 1 ] ] = fpose
        emin /= ( zeta ** -( 1.0 + globals()['FINE_LAMBDA'] ) )
        if emin < globals()['FINE_EMAX']:
          globals()['finepose'][ respond[ 1 ] ] += relpose
          reg = [ 'freg', globals()['nodeid'], \
          -globals()['finepose'][ respond[ 1 ] ] ]
          globals()['net'].send( respond[ 1 ], reg )
        else:
          del globals()['finepose'][ respond[ 1 ] ]
          print timestamp(), "Fine calibration error with node [", \
          respond[ 1 ], "] too large:", emin
      else:
        print timestamp(), "Not enough points shared with node [", \
        respond[ 1 ], "]."
    reg = [ 'freg' ]
    for node in globals()['group']:
      globals()['net'].send( node, reg )
    print timestamp(), "Stopping pose refinement registration process."
    globals()['net'].send( globals()['nodeid'], [ 'die' ] )


################################################################################
# FINE PROCESS 4: Pose Refinement Update
```

```
##############################################################################

class fine_update_thread( threading.Thread ):
  def run( self ):
    dc = 0
    while dc < len( globals()['net'] ) \
    or len( globals()['msgqueue'][ 'freg' ] ) > 0:
      # grab next message in the queue
      globals()['cvqueue'][ 'freg' ].acquire()
      while len( globals()['msgqueue'][ 'freg' ] ) == 0:
        globals()['cvqueue'][ 'freg' ].wait()
      reg = globals()['msgqueue'][ 'freg' ].pop( 0 )
      globals()['cvqueue'][ 'freg' ].release()
      if len( reg ) == 1:
        dc += 1
        continue
      # process fine registration
      print timestamp(), "Updating pairwise fine calibration with node [", \
      reg[ 1 ], "]."
      globals()['finepose'][ reg[ 1 ] ] = reg[ 2 ]
    print timestamp(), "Stopping pose refinement update process."
    globals()['net'].send( globals()['nodeid'], [ 'die' ] )


##############################################################################
# MAIN PROGRAM
##############################################################################

# check arguments
if len( sys.argv ) < 3:
  print "Usage:", sys.argv[ 0 ], "nodeid datadir"
  sys.exit( 1 )

print timestamp(), "Initializing node [", sys.argv[ 1 ], "]."
timestart = time.time()

# network initialization
net = network()
lines = csv.reader( open( sys.argv[ 2 ] + "/network" ) )
```

```
for line in lines:
  net.append( [ line[ 0 ], line[ 1 ], int( line[ 2 ] ), float( line[ 3 ] ), \
  float( line[ 4 ] ) ] )

# self initialization
nodeid = sys.argv[ 1 ]
groupid = nodeid
group = [ nodeid ]
coarsepose = { groupid : geometry.pose( 0, 0 ) }
finepose = { nodeid : geometry.pose( 0, 0 ) }
acc_matchdb = 0
acc_coarse = 0
acc_fine = 0

# point set intialization
points = []
lines = csv.reader( open( sys.argv[ 2 ] + "/" + nodeid + ".pts" ) )
for line in lines:
  points.append( geometry.point( float( line[ 0 ] ), float( line[ 1 ] ), \
  float( line[ 2 ] ) ) )
pointc = geometry.point( 0, 0, 0 )
for p in points:
  pointc += p
pointc /= float( len( points ) )

# message queue initialization
msgqueue = { 'cfeature':[], 'cmatch':[], 'cpose':[], 'cmerge':[], 'cack':[], \
'cupdate':[], 'finit':[], 'frespond':[], 'freg':[] }

cvqueue = { 'cfeature':threading.Condition(), 'cmatch':threading.Condition(), \
'cpose':threading.Condition(), 'cmerge':threading.Condition(), \
'cack':threading.Condition(), 'cupdate':threading.Condition(), \
'finit':threading.Condition(), 'frespond':threading.Condition(), \
'freg':threading.Condition() }

mergelock = threading.Lock()
groupupdate = threading.Event()

# start listening for connections
```

```
dscnode = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
dscnode.bind( net.hostport( nodeid ) )
dscnode.listen( 20 )

# start the various processing threads
print timestamp(), "Starting all threads."
threadcount = 10
coarse_feature_thread().start()
coarse_matching_thread().start()
coarse_matchproc_thread().start()
coarse_pose_thread().start()
coarse_merge_thread().start()
coarse_update_thread().start()
fine_init_thread().start()
fine_respond_thread().start()
fine_registration_thread().start()
fine_update_thread().start()

# add all incoming messages to the appropriate queue
inaccount = {}
while threadcount > 0:
  channel, details = dscnode.accept()
  inmsg = pickle.loads( channel.recv( 1048576 ) )
  channel.close()
  # bandwidth accounting
  t = int( time.time() )
  if inaccount.has_key( t ):
    inaccount[ t ] += len( pickle.dumps( inmsg ) )
  else:
    inaccount[ t ] = len( pickle.dumps( inmsg ) )
  if inmsg[ 0 ] == 'ping':
    continue
  elif inmsg[ 0 ] == 'die':
    threadcount -= 1
  else:
    cvqueue[ inmsg[ 0 ] ].acquire()
    msgqueue[ inmsg[ 0 ] ].append( inmsg )
    cvqueue[ inmsg[ 0 ] ].notify()
    cvqueue[ inmsg[ 0 ] ].release()
```

```
# close network
print timestamp(), "Closing socket for incoming messages."
dscnode.close()

# print basic output information
print timestamp(), "Calibration complete at node [", nodeid, "],", \
int( time.time() - timestart ), "seconds total."
print
print "Coarse Pose Estimate - Node [", nodeid, "], Group [", groupid, "]"
print coarsepose[ groupid ].T
print coarsepose[ groupid ].R
print
print "Fine pose estimates available for", len( finepose ), "nodes."

# interactive shell for additional output
while True:
  cmd = raw_input( "> " )
  if cmd == 'exit':
    break
  elif cmd == 'group':
    print "Group [", groupid, "] contains", group
  elif cmd == 'list':
    print finepose.keys()
  elif cmd == 'dump':
    pickle.dump( finepose, \
    open( sys.argv[ 2 ] + "/" + nodeid + ".fine", 'w' ) )
    print "Fine pose results for node [", nodeid, "] dumped."
  elif cmd and cmd.split()[ 0 ] == 'fine':
    if finepose.has_key( cmd.split()[ 1 ] ):
      print "Fine Pose Estimate - Node [", cmd.split()[ 1 ], "]"
      print finepose[ cmd.split()[ 1 ] ].T
      print finepose[ cmd.split()[ 1 ] ].R
    else:
      print "No fine pose estimate for [", cmd, "] exists."
  elif cmd == 'netstats':
    outbw = inbw = 0
    for i in net.outaccount.keys():
      outbw += net.outaccount[ i ]
```

```
    for i in inaccount.keys():
      inbw += inaccount[ i ]
    print "Bandwidth Usage Statistics - Node [", nodeid, "]"
    print "Total:", outbw, "bytes out,", inbw, "bytes in"
    print "Average:", ( outbw / ( max( net.outaccount.keys() ) - \
    min( net.outaccount.keys() ) ) ), "bytes/sec out,", ( inbw / ( \
    max( inaccount.keys() ) - min( inaccount.keys() ) ) ), "bytes/sec in"
    print "Peak:", max( net.outaccount.values() ), "bytes/sec out,", \
    max( inaccount.values() ), "bytes/sec in"
  elif cmd == 'netraw':
    print "OUT"
    for i in net.outaccount.keys():
      print str( i ) + "," + str( net.outaccount[ i ] )
    print "IN"
    for i in inaccount.keys():
      print str( i ) + "," + str( inaccount[ i ] )
  elif cmd == 'resources':
    print "Match Database Size:", acc_matchdb
    print "Coarse Registrations:", acc_coarse
    print "Fine Registrations:", acc_fine
  elif cmd == 'nrcsv':
    outbw = 0
    for i in net.outaccount.keys():
      outbw += net.outaccount[ i ]
    print outbw, acc_matchdb, acc_coarse, acc_fine
  else:
    print "Unrecognized command."
```

### B.1.2 Geometry Module

**geometry.py**

```
"""Geometry, point and pose classes"""
__author__ = 'Aaron Mavrinac'
__version__ = '1.0'

import math
import numpy

TOO_SMALL = 0.0000000001
```

```
class point:
  """3D point (vector) class"""
  def __init__( self, x, y, z = 0.0 ):
    self.x = float( x )
    self.y = float( y )
    self.z = float( z )
  def __getitem__( self, i ):
    if i == 0:
      return self.x
    elif i == 1:
      return self.y
    elif i == 2:
      return self.z
  def __setitem__( self, i, value ):
    if i == 0:
      self.x = value
    elif i == 1:
      self.y = value
    elif i == 2:
      self.z = value
  def __add__( self, p ):
    """Vector addition"""
    return point( self.x + p.x, self.y + p.y, self.z + p.z )
  def __sub__( self, p ):
    """Vector subtraction"""
    return point( self.x - p.x, self.y - p.y, self.z - p.z )
  def __mul__( self, p ):
    """Scalar multiplication or dot product"""
    if isinstance( p, point ):
      return ( self.x * p.x + self.y * p.y + self.z * p.z )
    else:
      return point( self.x * p, self.y * p, self.z * p )
  def __rmul__( self, p ):
    """Scalar multiplication or dot product"""
    return self.__mul__( p )
  def __div__( self, p ):
    """Scalar division"""
    return point( self.x / p, self.y / p, self.z / p )
```

```
  def __neg__( self ):
    """Negation"""
    return point( -self.x, -self.y, -self.z )
  def __repr__( self ):
    """Representation"""
    return "(" + str( self.x ) + ", " + str( self.y ) + ", " + \
    str( self.z ) + ")"
  def tuple( self ):
    """Returns the tuple of this vector"""
    return ( self.x, self.y, self.z )
  def array( self ):
    """Returns the NumPy array of this vector"""
    return numpy.array( [ [ self.x ], [ self.y ], [ self.z ] ] )
  def magnitude( self ):
    """Returns the magnitude of this vector"""
    return math.sqrt( self.x ** 2 + self.y ** 2 + self.z ** 2 )
  def normalize( self ):
    """Returns this vector normalized"""
    m = self.magnitude()
    return point( self.x / m, self.y / m, self.z / m )
  def euclidean( self, p ):
    """Returns the Euclidean distance to point p"""
    return math.sqrt( ( self.x - p.x ) ** 2 + ( self.y - p.y ) ** 2 + \
    ( self.z - p.z ) ** 2 )
  def angle( self, p ):
    """Returns the angle between this vector and vector p"""
    return math.fabs( math.acos( p.normalize() * self.normalize() ) )

class pose:
  """3D pose (rotation and translation) class"""
  def __init__( self, T, R ):
    if isinstance( T, point ):
      self.T = T
    else:
      self.T = point( 0, 0, 0 )
    if isinstance( R, numpy.ndarray ):
      self.R = R
    else:
      self.R = numpy.array( [ [ 1.0, 0.0, 0.0 ], [ 0.0, 1.0, 0.0 ], \
```

```
    [ 0.0, 0.0, 1.0 ] ] )
  def __add__( self, other ):
    """Pose composition: PB(PA(x)) = (PA + PB)(x)"""
    Tnew = point( ( other.R[ 0 ][ 0 ] * self.T.x + other.R[ 0 ][ 1 ] * \
    self.T.y + other.R[ 0 ][ 2 ] * self.T.z ), ( other.R[ 1 ][ 0 ] * \
    self.T.x + other.R[ 1 ][ 1 ] * self.T.y + other.R[ 1 ][ 2 ] * \
    self.T.z ), ( other.R[ 2 ][ 0 ] * self.T.x + other.R[ 2 ][ 1 ] * \
    self.T.y + other.R[ 2 ][ 2 ] * self.T.z ) ) + other.T
    Rnew = numpy.dot( other.R, self.R )
    return pose( Tnew, Rnew )
  def __sub__( self, other ):
    """Pose subtraction, inverts the right term and adds"""
    return self.__add__( -other )
  def __neg__( self ):
    """Pose inversion"""
    Rinv = self.R.transpose()
    Tinv = point( 0, 0, 0 )
    for i in range( 3 ):
      Tinv[ i ] = -( Rinv[ i ][ 0 ] * self.T.x + Rinv[ i ][ 1 ] * self.T.y + \
      Rinv[ i ][ 2 ] * self.T.z )
    return pose( Tinv, Rinv )
  def generate( self, x, y, z, theta, phi, psi ):
    """Generate a T and R given a translation point and 3 angles"""
    self.T = point( float( x ), float( y ), float( z ) )
    theta = float( theta ) % ( math.acos( -1 ) * 2 )
    phi = float( phi ) % ( math.acos( -1 ) * 2 )
    psi = float( psi ) % ( math.acos( -1 ) * 2 )
    self.R[ 0 ][ 0 ] = math.cos( phi ) * math.cos( psi )
    self.R[ 0 ][ 1 ] = math.sin( theta ) * math.sin( phi ) * \
    math.cos( psi ) - math.cos( theta ) * math.sin( psi )
    self.R[ 0 ][ 2 ] = math.cos( theta ) * math.sin( phi ) * \
    math.cos( psi ) + math.sin( theta ) * math.sin( psi )
    self.R[ 1 ][ 0 ] = math.cos( phi ) * math.sin( psi )
    self.R[ 1 ][ 1 ] = math.sin( theta ) * math.sin( phi ) * \
    math.sin( psi ) + math.cos( theta ) * math.cos( psi )
    self.R[ 1 ][ 2 ] = math.cos( theta ) * math.sin( phi ) * \
    math.sin( psi ) - math.sin( theta ) * math.cos( psi )
    self.R[ 2 ][ 0 ] = -math.sin( phi )
    self.R[ 2 ][ 1 ] = math.sin( theta ) * math.cos( phi )
```

```
    self.R[ 2 ][ 2 ] = math.cos( theta ) * math.cos( phi )
    for i in range( 3 ):
      for j in range( 3 ):
        if abs( self.R[ i ][ j ] ) < TOO_SMALL:
          self.R[ i ][ j ] = 0.0
  def om( self ):
    """Return the fixed-axis rotation angles from R"""
    phi = math.asin( -1.0 * self.R[ 2 ][ 0 ] )
    if phi < -TOO_SMALL:
      phi += 2.0 * math.pi
    if abs( phi ) < TOO_SMALL:
      phi = 0.0
    sign = math.cos( phi ) / abs( math.cos( phi ) )
    theta = math.atan( self.R[ 2 ][ 1 ] / self.R[ 2 ][ 2 ] )
    if abs( theta ) > TOO_SMALL:
      if sign * self.R[ 2 ][ 2 ] < 0:
        theta += math.pi
      elif sign * self.R[ 2 ][ 1 ] < 0:
        theta += 2.0 * math.pi
    else:
      theta = 0.0
    psi = math.atan( self.R[ 1 ][ 0 ] / self.R[ 0 ][ 0 ] )
    if abs( psi ) > TOO_SMALL:
      if sign * self.R[ 0 ][ 0 ] < 0:
        psi += math.pi
      elif sign * self.R[ 1 ][ 0 ] < 0:
        psi += 2.0 * math.pi
    else:
      psi = 0.0
    return ( theta, phi, psi )
  def nonzero( self ):
    """Check if this pose has any effect"""
    if self.T.x == 0 and self.T.y == 0 and self.T.z == 0 \
    and ( self.R - numpy.diag( numpy.array( [ 1, 1, 1 ] ) ) ).any() == 0:
      return False
    else:
      return True
  def map( self, p ):
    """Map a point through this pose"""
```

```
    return self.map_translate( self.map_rotate( p ) )
  def map_rotate( self, p ):
    """Rotation component of point mapping"""
    q = point( ( self.R[ 0 ][ 0 ] * p.x + self.R[ 0 ][ 1 ] * p.y + \
    self.R[ 0 ][ 2 ] * p.z ), ( self.R[ 1 ][ 0 ] * p.x + self.R[ 1 ][ 1 ] * \
    p.y + self.R[ 1 ][ 2 ] * p.z ), ( self.R[ 2 ][ 0 ] * p.x + \
    self.R[ 2 ][ 1 ] * p.y + self.R[ 2 ][ 2 ] * p.z ) )
    return q
  def map_translate( self, p ):
    """Translation component of point mapping"""
    q = p + self.T
    return q
```

### B.1.3 Registration Module

**registration.py**

```
"""Registration functions"""
__author__ = 'Aaron Mavrinac'
__version__ = '1.0'

import numpy
import geometry

def zeromatrix( m, n ):
  """Generates a matrix of m by n zeroes"""
  a = []
  for i in range( m ):
    a.append( [] )
    for j in range( n ):
      a[ i ].append( 0.0 )
  A = numpy.array( a )
  return A

def coarse_registration( M, P, tr ):
  """Coarse registration - DARCES (fully contained)"""
  rpose = geometry.pose( 0, 0 )
  # first control point
  for i in range( len( P ) ):
    for j in range( len( P ) ):
```

```
    if j == i:
      continue
    dps = P[ i ].euclidean( P[ j ] )
    # second control point
    for k in range( len( M ) ):
      for l in range( len( M ) ):
        if l == k:
          continue
        if abs( M[ k ].euclidean( M[ l ] ) - dps ) < tr:
  # third control point
          for m in range( len( P ) ):
            if m == i or m == j:
              continue
            dpa = P[ i ].euclidean( P[ m ] )
            dsa = P[ j ].euclidean( P[ m ] )
            for n in range( 0, len( M ) ):
              if n == k or n == l:
                continue
              if abs( M[ k ].euclidean( M[ n ] ) - dpa ) < tr \
              and abs( M[ l ].euclidean( M[ n ] ) - dsa ) < tr:
# calculate the optimal Euclidean transformation from M to P
                Pavg = geometry.point( 0, 0, 0 )
                Mavg = geometry.point( 0, 0, 0 )
                for pp in P:
                  Pavg += pp / float( len( P ) )
                for pm in M:
                  Mavg += pm / float( len( M ) )
                K = zeromatrix( 3, 3 )
                for pair in [ ( i, k ), ( j, l ), (m, n ) ]:
                  P[ pair[ 0 ] ] -= Pavg
                  M[ pair[ 1 ] ] -= Mavg
                  for x in range( 3 ):
                    for y in range( 3 ):
                      K[ x ][ y ] += P[ pair[ 0 ] ][ x ] * \
                      M[ pair[ 1 ] ][ y ]
                  P[ pair[ 0 ] ] += Pavg
                  M[ pair[ 1 ] ] += Mavg
                V, A, Ut = numpy.linalg.svd( K )
                vudet = numpy.linalg.det( numpy.dot( V, Ut ) )
```

```
                S = numpy.diag( numpy.array( [ 1, 1, vudet ] ) )
                R = numpy.dot( numpy.dot( V, S ), Ut )
                T = geometry.point( Pavg.x - ( R[ 0 ][ 0 ] * Mavg.x + \
                R[ 0 ][ 1 ] * Mavg.y + R[ 0 ][ 2 ] * Mavg.z ), Pavg.y - \
                ( R[ 1 ][ 0 ] * Mavg.x + R[ 1 ][ 1 ] * Mavg.y + \
                R[ 1 ][ 2 ] * Mavg.z ), Pavg.z - ( R[ 2 ][ 0 ] * Mavg.x + \
                R[ 2 ][ 1 ] * Mavg.y + R[ 2 ][ 2 ] * Mavg.z ) )
                rpose = geometry.pose( T, R )
  # verify the remaining points
                for pp in P:
                  f = False
                  for pm in M:
                    if abs( rpose.map( pm ).euclidean( pp ) ) < tr:
                      f = True
                      break
                  if not f:
                    rpose = geometry.pose( 0, 0 )
                    break
                if not f:
                  break
                return rpose
  return rpose

def fine_registration( M, S, zeta, te, tr ):
  """Fine registration - Trimmed Iterative Closest Point"""
  P = []
  for s in S:
    P.append( s )

  Slts = float( 10 ** 18 );
  Npo = int( zeta * len( P ) )
  e = Slts / float( Npo )
  pose = geometry.pose( 0, 0 )

  if Npo < 3:
    return pose, e

  for n in range( 100 ):
    # for each point in P, find closest in M and compute individual distances
```

```
pm = []
for pi in range( len( P ) ):
  pm.append( [ pi, 0, P[ pi ].euclidean( M[ 0 ] ) ] )
  for mi in range( 1, len( M ) ):
    if P[ pi ].euclidean( M[ mi ] ) < pm[ pi ][ 2 ]:
      pm[ pi ] = [ pi, mi, P[ pi ].euclidean( M[ mi ] ) ]

# sort distances in ascending order, select Npo least, compute Slts
pm.sort( lambda x, y : cmp( x[ 2 ], y[ 2 ] ) )
pm = pm[ :Npo ]
Slts = sum( x[ 2 ] ** 2 for x in pm )

# if any stop condition met, exit
ep = e
e = Slts / float( Npo )
if e < te or ( abs( e - ep ) / e ) < tr:
  break

# compute the optimal motion R,t minimizing Slts
Pavg = geometry.point( 0, 0, 0 )
Mavg = geometry.point( 0, 0, 0 )
for i in range( Npo ):
  Pavg += P[ pm[ i ][ 0 ] ] / float( Npo )
  Mavg += M[ pm[ i ][ 1 ] ] / float( Npo )
K = zeromatrix( 3, 3 )
for i in range( Npo ):
  K += P[ pm[ i ][ 0 ] ].array() * M[ pm[ i ][ 1 ] ].array().transpose()
K /= float( Npo )
K -= Pavg.array() * Mavg.array().transpose()
A = K - K.transpose()
B = K + K.transpose() - ( K.trace() * \
numpy.diag( numpy.array( [ 1.0, 1.0, 1.0 ] ) ) )
Q = zeromatrix( 4, 4 )
Q[ 0 ][ 0 ] = K.trace()
for i in range( 3 ):
  Q[ 0 ][ i + 1 ] = Q[ i + 1 ][ 0 ] = A[ ( i + 1 ) % 3 ][ ( i + 2 ) % 3 ]
  for j in range( 3 ):
    Q[ i + 1 ][ j + 1 ] = B[ i ][ j ]
w, v = numpy.linalg.eigh( Q )
```

```
    max = 0
    for i in range( 1, len( w ) ):
      if w[ i ] > w[ max ]:
        max = i
    qR = v[ :, max ]
    R = zeromatrix( 3, 3 )
    R[ 0 ][ 0 ] = qR[ 0 ] ** 2 + qR[ 1 ] ** 2 - qR[ 2 ] ** 2 - qR[ 3 ] ** 2
    R[ 0 ][ 1 ] = 2 * ( qR[ 1 ] * qR[ 2 ] - qR[ 0 ] * qR[ 3 ] )
    R[ 0 ][ 2 ] = 2 * ( qR[ 1 ] * qR[ 3 ] + qR[ 0 ] * qR[ 2 ] )
    R[ 1 ][ 0 ] = 2 * ( qR[ 1 ] * qR[ 2 ] + qR[ 0 ] * qR[ 3 ] )
    R[ 1 ][ 1 ] = qR[ 0 ] ** 2 + qR[ 2 ] ** 2 - qR[ 1 ] ** 2 - qR[ 3 ] ** 2
    R[ 1 ][ 2 ] = 2 * ( qR[ 2 ] * qR[ 3 ] - qR[ 0 ] * qR[ 1 ] )
    R[ 2 ][ 0 ] = 2 * ( qR[ 1 ] * qR[ 3 ] - qR[ 0 ] * qR[ 2 ] )
    R[ 2 ][ 1 ] = 2 * ( qR[ 2 ] * qR[ 3 ] + qR[ 0 ] * qR[ 1 ] )
    R[ 2 ][ 2 ] = qR[ 0 ] ** 2 + qR[ 3 ] ** 2 - qR[ 1 ] ** 2 - qR[ 2 ] ** 2
    RP = numpy.dot( R, numpy.array( [ [ Pavg.x ], [ Pavg.y ], [ Pavg.z ] ] ) )
    T = Mavg - geometry.point( RP[ 0 ], RP[ 1 ], RP[ 2 ] )
    qpose = geometry.pose( T, R )

    # transform P by R,t and loop
    for i in range( len( P ) ):
      P[ i ] = qpose.map( P[ i ] )

    pose += qpose

  return pose, e
```

## B.2 Local Point Detection (C)

### B.2.1 Main Program

**capture.c**

```
#include <userint.h>
#include <ansi_c.h>
#include <cvirte.h>
#include "nivision.h"
#include "vp_imaq.h"
#include "fast.h"
```

```
#include "stereo.h"
#include "capture.h"
#include <formatio.h>

//#define DBG_WRITE_BMP
//#define DBG_SHOW_ITER_NUM
//#define DBG_POINT_ACCEPT
//#define DBG_CORR_VERIFY
//#define DBG_DISABLE_PYDSC

#define MANUAL_CORNERS 20
#define FAST_PROX 5
#define FAST_MINTHRESH 10
#define CORR_WINSIZE 13
#define CORR_YTHRESH 0.005
#define CORR_THRESH 0.9
#define TRI_ZMIN 100.0
#define TRI_ZMAX 2000.0
#define NONMAX 1

static int mainpanel;

int main (int argc, char *argv[])
{
  if (InitCVIRTE (0, argv, 0) == 0)
    return -1;  /* out of memory */
  if ((mainpanel = LoadPanel (0, "capture.uir", MAINPANEL)) < 0)
    return -1;
  DisplayPanel (mainpanel);
  RunUserInterface ();
  DiscardPanel (mainpanel);
  return 0;
}

int CVICALLBACK mp_cb (int panel, int event, void *callbackData,
    int eventData1, int eventData2)
{
  switch (event)
  {
```

```
    case EVENT_GOT_FOCUS:

      break;
    case EVENT_LOST_FOCUS:

      break;
    case EVENT_CLOSE:
      exit( 0 );
      break;
  }
  return 0;
}

int CVICALLBACK capture (int panel, int control, int event,
    void *callbackData, int eventData1, int eventData2)
{
  int i, j, x, y, SL, SR, linked, manual, thresh, threshinc, np, pmin, pmax,
      width, height, ncl, ncr, nclnm, ncrnm, win_r, zncc_maxj, fH;
  char nodeid[ 256 ], ldev[ 32 ], rdev[ 32 ], point[ 256 ];
  double lpar[ 9 ];
  double rpar[ 9 ];
  double T[ 3 ], om[ 3 ];
  double ** R, ** E;
  double epi[ 3 ];
  int * P;
  xy * CLraw, * CRraw;
  Point * CL, * CR;
  dxy * CLn, * CRn;
  dxy CRn_rect;
  Image * SnapL, * SnapR, * ImgL, * ImgR;
  unsigned char * lim, * rim;
  double Lcorr[ CORR_WINSIZE ][ CORR_WINSIZE ],
         Rcorr[ CORR_WINSIZE ][ CORR_WINSIZE ];
  double winarea, Lavg, Ravg,
         zncc, zncc_a, zncc_b, zncc_top, zncc_boa, zncc_bob;
  double * zncc_max;
  dxyz pt3d;
  WindowEventType man_event;
  Rect man_rect;
```

```
  /* debug variables */
  RGBValue ovlc;
  dxy dbgdxy;
  Point dbgpt;
  int corrpct, corrtot, flag, pole;

  switch (event)
  {
    case EVENT_COMMIT:

/* === PARAMETERS & INITIALIZATION === */

      SetCtrlVal( panel, MAINPANEL_LED_CPU, TRUE );
      DisplayPanel( panel );

      GetCtrlVal( panel, MAINPANEL_STR_NODEID, nodeid );
      GetCtrlVal( panel, MAINPANEL_STR_CAML_DEV, ldev );
      GetCtrlVal( panel, MAINPANEL_STR_CAMR_DEV, rdev );
      GetCtrlVal( panel, MAINPANEL_NUM_CAML_FC1, &lpar[ 0 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_CAML_FC2, &lpar[ 1 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_CAML_CC1, &lpar[ 2 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_CAML_CC2, &lpar[ 3 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_CAML_KC1, &lpar[ 4 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_CAML_KC2, &lpar[ 5 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_CAML_KC3, &lpar[ 6 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_CAML_KC4, &lpar[ 7 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_CAML_KC5, &lpar[ 8 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_CAMR_FC1, &rpar[ 0 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_CAMR_FC2, &rpar[ 1 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_CAMR_CC1, &rpar[ 2 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_CAMR_CC2, &rpar[ 3 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_CAMR_KC1, &rpar[ 4 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_CAMR_KC2, &rpar[ 5 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_CAMR_KC3, &rpar[ 6 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_CAMR_KC4, &rpar[ 7 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_CAMR_KC5, &rpar[ 8 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_STEREO_T1, &T[ 0 ] );
      GetCtrlVal( panel, MAINPANEL_NUM_STEREO_T2, &T[ 1 ] );
```

```
    GetCtrlVal( panel, MAINPANEL_NUM_STEREO_T3, &T[ 2 ] );
    GetCtrlVal( panel, MAINPANEL_NUM_STEREO_OM1, &om[ 0 ] );
    GetCtrlVal( panel, MAINPANEL_NUM_STEREO_OM2, &om[ 1 ] );
    GetCtrlVal( panel, MAINPANEL_NUM_STEREO_OM3, &om[ 2 ] );

    GetCtrlVal( panel, MAINPANEL_BIN_LINKED, &linked );
    GetCtrlVal( panel, MAINPANEL_BIN_FAST_MANUAL, &manual );

    ovlc.R = ovlc.G = ovlc.B = 255;
    corrpct = corrtot = flag = 0;

    imaqSetWindowThreadPolicy( IMAQ_SEPARATE_THREAD );
    for( i = 0; i < 2; i++ )
      imaqShowScrollbars( i, TRUE );

    /* compute the rotation matrix and essential matrix */

    R = stereo_rodrigues( om[ 0 ], om[ 1 ], om[ 2 ] );
    E = stereo_essential( om[ 0 ], om[ 1 ], om[ 2 ], T );


/* === IMAGE ACQUISITION === */

    /* prepare image buffers */

    SnapL = imaqCreateImage( IMAQ_IMAGE_U8, 0 );
    SnapR = imaqCreateImage( IMAQ_IMAGE_U8, 0 );
    ImgL = imaqCreateImage( IMAQ_IMAGE_U8, 0 );
    ImgR = imaqCreateImage( IMAQ_IMAGE_U8, 0 );

    /* capture the images */

    SetCtrlVal( panel, MAINPANEL_LED_1394, TRUE );
    DisplayPanel( panel );
    if( linked )
    {
      SL = vp_imaq_open( ldev );
      SR = vp_imaq_open( rdev );
      vp_imaq_snap_stereo( SL, SR, SnapL, SnapR );
```

```
        vp_imaq_close( SL );
        vp_imaq_close( SR );
      }
      else
      {
        SL = vp_imaq_open( ldev );
        vp_imaq_snap( SL, SnapL );
        vp_imaq_close( SL );
        SR = vp_imaq_open( rdev );
        vp_imaq_snap( SR, SnapR );
        vp_imaq_close( SR );
      }
      SetCtrlVal( panel, MAINPANEL_LED_1394, FALSE );
      DisplayPanel( panel );

      /* convert to grayscale */

      imaqCast( ImgL, SnapL, IMAQ_IMAGE_U8, NULL, 8 );
      imaqCast( ImgR, SnapR, IMAQ_IMAGE_U8, NULL, 8 );

      /* debug: write out BMP files */

      #ifdef DBG_WRITE_BMP
      sprintf( point, "%s-left.bmp", nodeid );
      imaqWriteBMPFile( ImgL, point, FALSE, NULL );
      sprintf( point, "%s-right.bmp", nodeid );
      imaqWriteBMPFile( ImgR, point, FALSE, NULL );
      #endif


/* === INTEREST POINT DETECTION === */

      if( manual )
      {
        /* manual interest point selection */

        ncl = ncr = MANUAL_CORNERS;

        CL = ( Point * )malloc( ncl * sizeof( Point ) );
```

```
CR = ( Point * )malloc( ncr * sizeof( Point ) );
P = ( int * )malloc( ncl * sizeof( int ) );

imaqDisplayImage( ImgL, 0, FALSE );
imaqDisplayImage( ImgR, 1, FALSE );

imaqSetCurrentTool( IMAQ_POINT_TOOL );

i = 0;
while( i < MANUAL_CORNERS )
{
  j = 0;
  while( j < 3 )
  {
    imaqGetLastEvent( &man_event, &x, NULL, &man_rect );
    if( man_event == IMAQ_CLICK_EVENT && x == 0 )
    {
      CL[ i ] = imaqMakePoint( man_rect.left, man_rect.top );
      j |= 1;
    }
    if( man_event == IMAQ_CLICK_EVENT && x == 1 )
    {
      CR[ i ] = imaqMakePoint( man_rect.left, man_rect.top );
      j |= 2;
    }
  }
  if( ConfirmPopup( "Point Detection", "Accept this pair?" ) )
  {
    imaqOverlayPoints( ImgL, &CL[ i ], 1, &ovlc, 1,
              IMAQ_POINT_AS_CROSS, NULL, NULL );
    imaqOverlayPoints( ImgR, &CR[ i ], 1, &ovlc, 1,
              IMAQ_POINT_AS_CROSS, NULL, NULL );
    imaqDisplayImage( ImgL, 0, FALSE );
    imaqDisplayImage( ImgR, 1, FALSE );
    P[ i ] = i;
    i++;
  }
}
```

```
    MessagePopup( "Point Detection", "Manual point selection complete." );
    imaqSetCurrentTool( IMAQ_NO_TOOL );

    /* normalize the interest point coordinates */

    CLn = ( dxy * )malloc( ncl * sizeof( dxy ) );
    CRn = ( dxy * )malloc( ncr * sizeof( dxy ) );
    for( i = 0; i < ncl; i++ )
      stereo_normalize( CL[ i ].x, CL[ i ].y, lpar[ 0 ], lpar[ 1 ],
                lpar[ 2 ], lpar[ 3 ], lpar[ 4 ], lpar[ 5 ],
                        lpar[ 6 ], lpar[ 7 ], lpar[ 8 ], &CLn[ i ] );
    for( i = 0; i < ncr; i++ )
      stereo_normalize( CR[ i ].x, CR[ i ].y, rpar[ 0 ], rpar[ 1 ],
                rpar[ 2 ], rpar[ 3 ], rpar[ 4 ], rpar[ 5 ],
                        rpar[ 6 ], rpar[ 7 ], rpar[ 8 ], &CRn[ i ] );
  }
  else
  {
    win_r = ( CORR_WINSIZE - 1 ) / 2;
    winarea = ( double )( 4 * win_r * win_r + 4 * win_r + 1 );

    lim = imaqImageToArray( ImgL, IMAQ_NO_RECT, &width, &height );
    rim = imaqImageToArray( ImgR, IMAQ_NO_RECT, NULL, NULL );

    GetCtrlVal( panel, MAINPANEL_NUM_FAST_THRESH, &thresh );
    threshinc = thresh / 2;
    GetCtrlVal( panel, MAINPANEL_NUM_PTS_BASE, &pmin );
    GetCtrlVal( panel, MAINPANEL_NUM_PTS_VAR, &pmax );
    pmin -= pmax;
    pmax *= 2;
    pmax += pmin;

    CL = NULL;
    CR = NULL;
    P = NULL;
    CLn = NULL;
    CRn = NULL;
    zncc_max = NULL;
```

```
while( 1 )
{
  /* FAST corner detection */

  CLraw = fast_corner_detect_9( lim, width, height, thresh, &ncl );
  if( NONMAX )
  {
    CLraw = fast_nonmax( lim, width, height, CLraw, ncl, thresh,
                         &nclnm );
    ncl = nclnm;
  }

  CRraw = fast_corner_detect_9( rim, width, height, thresh, &ncr );
  if( NONMAX )
  {
    CRraw = fast_nonmax( rim, width, height, CRraw, ncr, thresh,
                         &ncrnm );
    ncr = ncrnm;
  }

  /* clean up the detected corners */

  j = 0;
  for( i = 0; i < ncl; i++ )
  {
    if( ! ( CLraw[ i ].x > win_r
        && CLraw[ i ].x < ( width - win_r - 1 )
        && CLraw[ i ].y > win_r
        && CLraw[ i ].y < ( height - win_r - 1 ) ) )
      CLraw[ i ].x = -1;
    for( x = 0; x < i; x++ )
      if( CLraw[ x ].x < ( CLraw[ i ].x + FAST_PROX )
          && CLraw[ x ].x > ( CLraw[ i ].x - FAST_PROX )
          && CLraw[ x ].y <= CLraw[ i ].y
          && CLraw[ x ].y > ( CLraw[ i ].y - FAST_PROX ) )
        CLraw[ i ].x = -1;
    if( CLraw[ i ].x >= 0 )
      j++;
  }
```

```
CL = ( Point * )realloc( CL, j * sizeof( Point ) );
j = 0;
for( i = 0; i < ncl; i++ )
  if( CLraw[ i ].x >= 0 )
    CL[ j++ ] = imaqMakePoint( CLraw[ i ].x, CLraw[ i ].y );
ncl = j;
j = 0;
for( i = 0; i < ncr; i++ )
{
  if( ! ( CRraw[ i ].x > win_r
      && CRraw[ i ].x < ( width - win_r - 1 )
      && CRraw[ i ].y > win_r
      && CRraw[ i ].y < ( height - win_r - 1 ) ) )
    CRraw[ i ].x = -1;
  for( x = 0; x < i; x++ )
    if( CRraw[ x ].x < ( CRraw[ i ].x + FAST_PROX )
        && CRraw[ x ].x > ( CRraw[ i ].x - FAST_PROX )
        && CRraw[ x ].y <= CRraw[ i ].y
        && CRraw[ x ].y > ( CRraw[ i ].y - FAST_PROX ) )
      CRraw[ i ].x = -1;
  if( CRraw[ i ].x >=0 )
    j++;
}
CR = ( Point * )realloc( CR, j * sizeof( Point ) );
j = 0;
for( i = 0; i < ncr; i++ )
  if( CRraw[ i ].x >= 0 )
    CR[ j++ ] = imaqMakePoint( CRraw[ i ].x, CRraw[ i ].y );
ncr = j;

P = ( int * )realloc( P, ncl * sizeof( int ) );

/* normalize the interest point coordinates */

CLn = ( dxy * )realloc( CLn, ncl * sizeof( dxy ) );
CRn = ( dxy * )realloc( CRn, ncr * sizeof( dxy ) );
for( i = 0; i < ncl; i++ )
  stereo_normalize( CL[ i ].x, CL[ i ].y, lpar[ 0 ], lpar[ 1 ],
                    lpar[ 2 ], lpar[ 3 ], lpar[ 4 ], lpar[ 5 ],
```

```
                    lpar[ 6 ], lpar[ 7 ], lpar[ 8 ], &CLn[ i ] );
for( i = 0; i < ncr; i++ )
  stereo_normalize( CR[ i ].x, CR[ i ].y, rpar[ 0 ], rpar[ 1 ],
                    rpar[ 2 ], rpar[ 3 ], rpar[ 4 ], rpar[ 5 ],
                    rpar[ 6 ], rpar[ 7 ], rpar[ 8 ], &CRn[ i ] );

/* epipolar-constrained ZNCC correspondence */

zncc_max = ( double * )realloc( zncc_max, ncl * sizeof( double ) );
for( i = 0; i < ncl; i++ )
{
  Lavg = 0.0;
  for( x = -win_r; x <= win_r; x++ )
  for( y = -win_r; y <= win_r; y++ )
  {
    Lcorr[ x + win_r ][ y + win_r ] =
      ( double )lim[ CL[ i ].x + x + width * ( CL[ i ]. y + y ) ];
    Lavg += Lcorr[ x + win_r ][ y + win_r ];
  }
  Lavg /= winarea;

  zncc_max[ i ] = CORR_THRESH;
  zncc_maxj = -1;

  for( j = 0; j < 3; j++ )
    epi[ j ] = E[ j ][ 0 ] * CLn[ i ].x + E[ j ][ 1 ] * CLn[ i ].y
               + E[ j ][ 2 ];

  for( j = 0; j < ncr; j++ )
  {
    if( fabs( ( epi[ 0 ] * CRn[ j ].x + epi[ 1 ] * CRn[ j ].y
              + epi[ 2 ] ) / sqrt( epi[ 0 ] * epi[ 0 ]
              + epi[ 1 ] * epi[ 1 ] ) ) < CORR_YTHRESH )
    {
      Ravg = 0.0;
      for( x = -win_r; x <= win_r; x++ )
      for( y = -win_r; y <= win_r; y++ )
      {
        Rcorr[ x + win_r ][ y + win_r ] =
```

```
            ( double )rim[ CR[ j ].x + x + width * ( CR[ j ]. y + y ) ];
          Ravg += Rcorr[ x + win_r ][ y + win_r ];
        }
        Ravg /= winarea;

        zncc_top = zncc_boa = zncc_bob = 0.0;
        for( x = 0; x < CORR_WINSIZE; x++ )
        for( y = 0; y < CORR_WINSIZE; y++ )
        {
          zncc_a = ( Lcorr[ x ][ y ] - Lavg );
          zncc_b = ( Rcorr[ x ][ y ] - Ravg );
          zncc_top += zncc_a * zncc_b;
          zncc_boa += zncc_a * zncc_a;
          zncc_bob += zncc_b * zncc_b;
        }
        zncc = zncc_top / sqrt( zncc_boa * zncc_bob );

        if( zncc > zncc_max[ i ] )
        {
          zncc_maxj = j;
          zncc_max[ i ] = zncc;
        }
      }
    }

    P[ i ] = zncc_maxj;
    if( zncc_maxj > -1 )
      for( j = 0; j < i; j++ )
        if( P[ j ] == zncc_maxj )
        {
          if( zncc_max[ j ] > zncc_max[ i ] )
          {
            P[ i ] = -1;
            break;
          }
          else
            P[ j ] = -1;
        }
  }
```

```
    /* change thresh */
    np = 0;
    for( i = 0; i < ncl; i++ )
      if( P[ i ] > -1 )
        np++;

    #ifdef DBG_SHOW_ITER_NUM
    sprintf( point, "Detected %d points using threshold %d.",
             np, thresh );
    MessagePopup( "Point Detection", point );
    #endif

    if( np < pmin )
    {
      if( threshinc > 0 )
        threshinc /= -2;
    }
    else if( np > pmax )
    {
      if( threshinc < 0 )
        threshinc /= -2;
    }
    else
      break;
    if( ! threshinc )
      break;
    thresh += threshinc;
    if( thresh < FAST_MINTHRESH )
    {
      thresh = FAST_MINTHRESH;
      threshinc = 0;
    }
  }
}

/* interest point verification */

if( ncl > 0 && ncr > 0 )
```

```
    {
      #ifdef DBG_POINT_ACCEPT
      if( ! manual )
      {
        for( i = 0; i < ncl; i++ )
        {
          if( P[ i ] == -1 )
            continue;
          imaqOverlayPoints( ImgL, &CL[ i ], 1, &ovlc, 1,
                             IMAQ_POINT_AS_CROSS, NULL, NULL );
          imaqOverlayPoints( ImgR, &CR[ P[ i ] ], 1, &ovlc, 1,
                             IMAQ_POINT_AS_CROSS, NULL, NULL );
        }
        imaqDisplayImage( ImgL, 0, FALSE );
        imaqDisplayImage( ImgR, 1, FALSE );
        if( ! ConfirmPopup( "Point Detection",
            "Accept these interest points?" ) )
          break;
      }
      #endif
    }
    else
    {
      MessagePopup( "Point Detection", "No interest points detected." );
      break;
    }


/* === TRIANGULATION === */

    sprintf( point, "..\\pydsc\\experiment\\demo\\%s.pts", nodeid );
    fH = OpenFile( point, VAL_WRITE_ONLY, VAL_TRUNCATE, VAL_ASCII );

    for( i = 0; i < ncl; i++ )
    {
      if( P[ i ] == -1 )
        continue;

      /* debug: interactive verification */
```

```
        #ifdef DBG_CORR_VERIFY
        imaqClearOverlay( ImgL, NULL );
        imaqClearOverlay( ImgR, NULL );
        imaqOverlayPoints( ImgL, &CL[ i ], 1, &ovlc, 1,
                           IMAQ_POINT_AS_CROSS, NULL, NULL );
        imaqOverlayPoints( ImgR, &CR[ P[ i ] ], 1, &ovlc, 1,
                           IMAQ_POINT_AS_CROSS, NULL, NULL );
        imaqDisplayImage( ImgL, 0, FALSE );
        imaqDisplayImage( ImgR, 1, FALSE );
        corrtot++;
        if( ! ConfirmPopup( "Point Detection", "Do these points match?" ) )
          continue;
        else
          corrpct++;
        #endif

        pt3d = stereo_triangulate( CLn[ i ], CRn[ P[ i ] ], R, T );

        if( pt3d.z > TRI_ZMIN && pt3d.z < TRI_ZMAX )
        {
          sprintf( point, "%lf,%lf,%lf", pt3d.x, pt3d.y, pt3d.z );
          WriteLine( fH, point, -1 );
        }
      }

      #ifdef DBG_CORR_VERIFY
      sprintf( point, "Correspondence matched %.2f%% correctly.",
              ( 100.0 * ( float )corrpct / ( float )corrtot ) );
      MessagePopup( "Point Detection", point );
      #endif

      CloseFile( fH );


/* === CLEANUP === */

      free( CL );
      free( CR );
```

```
      free( CLn );
      free( CRn );
      free( P );
      if( ! manual )
        free( zncc_max );
      free( R );
      free( E );

      SetCtrlVal( panel, MAINPANEL_LED_CPU, FALSE );

/* === START PYDSC CALIBRATION === */

      #ifndef DBG_DISABLE_PYDSC
      sprintf( point, "..\\pydsc\\pydsc_demo_%s.bat", nodeid );
      system( point );
      #endif

      break;
  }
  return 0;
}

int CVICALLBACK calibrate (int panel, int control, int event,
    void *callbackData, int eventData1, int eventData2)
{
  int i, num_images, SL, SR, linked;
  char nodeid[ 256 ], ldev[ 32 ], rdev[ 32 ];
  char fn[ 256 ];
  Image * SnapL, * SnapR, * ImgL, * ImgR;

  switch (event)
  {
    case EVENT_COMMIT:
      GetCtrlVal( panel, MAINPANEL_NUM_CAL_IMAGES, &num_images );
      GetCtrlVal( panel, MAINPANEL_STR_NODEID, nodeid );
      GetCtrlVal( panel, MAINPANEL_STR_CAML_DEV, ldev );
      GetCtrlVal( panel, MAINPANEL_STR_CAMR_DEV, rdev );
      GetCtrlVal( panel, MAINPANEL_BIN_LINKED, &linked );
```

```
/* prepare image buffers */

SnapL = imaqCreateImage( IMAQ_IMAGE_U8, 0 );
SnapR = imaqCreateImage( IMAQ_IMAGE_U8, 0 );
ImgL = imaqCreateImage( IMAQ_IMAGE_U8, 0 );
ImgR = imaqCreateImage( IMAQ_IMAGE_U8, 0 );

for( i = 0; i < 2; i++ )
  imaqShowScrollbars( i, TRUE );

for( i = 1; i <= num_images; i++ )
{
  /* capture the images */

  if( linked )
  {
    SL = vp_imaq_open( ldev );
    SR = vp_imaq_open( rdev );
    vp_imaq_snap_stereo( SL, SR, SnapL, SnapR );
    vp_imaq_close( SL );
    vp_imaq_close( SR );
  }
  else
  {
    SL = vp_imaq_open( ldev );
    vp_imaq_snap( SL, SnapL );
    vp_imaq_close( SL );
    SR = vp_imaq_open( rdev );
    vp_imaq_snap( SR, SnapR );
    vp_imaq_close( SR );
  }

  /* convert to grayscale */

  imaqCast( ImgL, SnapL, IMAQ_IMAGE_U8, NULL, 8 );
  imaqCast( ImgR, SnapR, IMAQ_IMAGE_U8, NULL, 8 );

  /* output to files */
```

```
        sprintf( fn, "%s-left%.2d.bmp", nodeid, i );
        imaqWriteBMPFile( ImgL, fn, FALSE, NULL );
        sprintf( fn, "%s-right%.2d.bmp", nodeid, i );
        imaqWriteBMPFile( ImgR, fn, FALSE, NULL );

        /* display images and ask for approval */

        imaqDisplayImage( ImgL, 0, FALSE );
        imaqDisplayImage( ImgR, 1, FALSE );

        if( ! ConfirmPopup( "Camera Calibration", "Accept this image pair?" ) )
            i--;
      }

      MessagePopup( "Camera Calibration", "Image acquisition complete." );

      break;
  }
  return 0;
}

void CVICALLBACK param_load (int menuBar, int menuItem, void *callbackData,
                             int panel)
{
  int fH;
  char file[ MAX_PATHNAME_LEN ], data[ 4096 ],
       nodeid[ 256 ], ldev[ 32 ], rdev[ 32 ];
  double lfc1, lfc2, lcc1, lcc2, lkc1, lkc2, lkc3, lkc4, lkc5;
  double rfc1, rfc2, rcc1, rcc2, rkc1, rkc2, rkc3, rkc4, rkc5;
  double st1, st2, st3, som1, som2, som3;

  if( FileSelectPopup( "", "*.dsc", "", "Load Node Parameters",
      VAL_LOAD_BUTTON, 0, 1, 1, 0, file ) == 1 )
  {
    fH = OpenFile( file, VAL_READ_ONLY, NULL, VAL_ASCII );
    ReadLine( fH, nodeid, 255 );
    ReadLine( fH, ldev, 31 );
    ReadLine( fH, rdev, 31 );
    ReadLine( fH, data, 4095 );
```

```
    sscanf( data, "%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,
                   %lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf",
        &lfc1, &lfc2, &lcc1, &lcc2, &lkc1, &lkc2, &lkc3, &lkc4, &lkc5,
        &rfc1, &rfc2, &rcc1, &rcc2, &rkc1, &rkc2, &rkc3, &rkc4, &rkc5,
        &st1, &st2, &st3, &som1, &som2, &som3 );
    CloseFile( fH );

    SetCtrlVal( panel, MAINPANEL_STR_NODEID, nodeid );
    SetCtrlVal( panel, MAINPANEL_STR_CAML_DEV, ldev );
    SetCtrlVal( panel, MAINPANEL_STR_CAMR_DEV, rdev );
    SetCtrlVal( panel, MAINPANEL_NUM_CAML_FC1, lfc1 );
    SetCtrlVal( panel, MAINPANEL_NUM_CAML_FC2, lfc2 );
    SetCtrlVal( panel, MAINPANEL_NUM_CAML_CC1, lcc1 );
    SetCtrlVal( panel, MAINPANEL_NUM_CAML_CC2, lcc2 );
    SetCtrlVal( panel, MAINPANEL_NUM_CAML_KC1, lkc1 );
    SetCtrlVal( panel, MAINPANEL_NUM_CAML_KC2, lkc2 );
    SetCtrlVal( panel, MAINPANEL_NUM_CAML_KC3, lkc3 );
    SetCtrlVal( panel, MAINPANEL_NUM_CAML_KC4, lkc4 );
    SetCtrlVal( panel, MAINPANEL_NUM_CAML_KC5, lkc5 );
    SetCtrlVal( panel, MAINPANEL_NUM_CAMR_FC1, rfc1 );
    SetCtrlVal( panel, MAINPANEL_NUM_CAMR_FC2, rfc2 );
    SetCtrlVal( panel, MAINPANEL_NUM_CAMR_CC1, rcc1 );
    SetCtrlVal( panel, MAINPANEL_NUM_CAMR_CC2, rcc2 );
    SetCtrlVal( panel, MAINPANEL_NUM_CAMR_KC1, rkc1 );
    SetCtrlVal( panel, MAINPANEL_NUM_CAMR_KC2, rkc2 );
    SetCtrlVal( panel, MAINPANEL_NUM_CAMR_KC3, rkc3 );
    SetCtrlVal( panel, MAINPANEL_NUM_CAMR_KC4, rkc4 );
    SetCtrlVal( panel, MAINPANEL_NUM_CAMR_KC5, rkc5 );
    SetCtrlVal( panel, MAINPANEL_NUM_STEREO_T1, st1 );
    SetCtrlVal( panel, MAINPANEL_NUM_STEREO_T2, st2 );
    SetCtrlVal( panel, MAINPANEL_NUM_STEREO_T3, st3 );
    SetCtrlVal( panel, MAINPANEL_NUM_STEREO_OM1, som1 );
    SetCtrlVal( panel, MAINPANEL_NUM_STEREO_OM2, som2 );
    SetCtrlVal( panel, MAINPANEL_NUM_STEREO_OM3, som3 );
  }
}

void CVICALLBACK param_save (int menuBar, int menuItem, void *callbackData,
    int panel)
```

```
{
  int fH;
  char file[ MAX_PATHNAME_LEN ], data[ 4096 ],
       nodeid[ 256 ], ldev[ 32 ], rdev[ 32 ];
  double lfc1, lfc2, lcc1, lcc2, lkc1, lkc2, lkc3, lkc4, lkc5;
  double rfc1, rfc2, rcc1, rcc2, rkc1, rkc2, rkc3, rkc4, rkc5;
  double st1, st2, st3, som1, som2, som3;

  if( FileSelectPopup( "", "*.dsc", "", "Save Node Parameters",
      VAL_SAVE_BUTTON, 0, 1, 1, 1, file ) )
  {
    GetCtrlVal( panel, MAINPANEL_STR_NODEID, nodeid );
    GetCtrlVal( panel, MAINPANEL_STR_CAML_DEV, ldev );
    GetCtrlVal( panel, MAINPANEL_STR_CAMR_DEV, rdev );
    GetCtrlVal( panel, MAINPANEL_NUM_CAML_FC1, &lfc1 );
    GetCtrlVal( panel, MAINPANEL_NUM_CAML_FC2, &lfc2 );
    GetCtrlVal( panel, MAINPANEL_NUM_CAML_CC1, &lcc1 );
    GetCtrlVal( panel, MAINPANEL_NUM_CAML_CC2, &lcc2 );
    GetCtrlVal( panel, MAINPANEL_NUM_CAML_KC1, &lkc1 );
    GetCtrlVal( panel, MAINPANEL_NUM_CAML_KC2, &lkc2 );
    GetCtrlVal( panel, MAINPANEL_NUM_CAML_KC3, &lkc3 );
    GetCtrlVal( panel, MAINPANEL_NUM_CAML_KC4, &lkc4 );
    GetCtrlVal( panel, MAINPANEL_NUM_CAML_KC5, &lkc5 );
    GetCtrlVal( panel, MAINPANEL_NUM_CAMR_FC1, &rfc1 );
    GetCtrlVal( panel, MAINPANEL_NUM_CAMR_FC2, &rfc2 );
    GetCtrlVal( panel, MAINPANEL_NUM_CAMR_CC1, &rcc1 );
    GetCtrlVal( panel, MAINPANEL_NUM_CAMR_CC2, &rcc2 );
    GetCtrlVal( panel, MAINPANEL_NUM_CAMR_KC1, &rkc1 );
    GetCtrlVal( panel, MAINPANEL_NUM_CAMR_KC2, &rkc2 );
    GetCtrlVal( panel, MAINPANEL_NUM_CAMR_KC3, &rkc3 );
    GetCtrlVal( panel, MAINPANEL_NUM_CAMR_KC4, &rkc4 );
    GetCtrlVal( panel, MAINPANEL_NUM_CAMR_KC5, &rkc5 );
    GetCtrlVal( panel, MAINPANEL_NUM_STEREO_T1, &st1 );
    GetCtrlVal( panel, MAINPANEL_NUM_STEREO_T2, &st2 );
    GetCtrlVal( panel, MAINPANEL_NUM_STEREO_T3, &st3 );
    GetCtrlVal( panel, MAINPANEL_NUM_STEREO_OM1, &som1 );
    GetCtrlVal( panel, MAINPANEL_NUM_STEREO_OM2, &som2 );
    GetCtrlVal( panel, MAINPANEL_NUM_STEREO_OM3, &som3 );
```

```
    sprintf( data, "%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,
                    %lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf",
          lfc1, lfc2, lcc1, lcc2, lkc1, lkc2, lkc3, lkc4, lkc5,
          rfc1, rfc2, rcc1, rcc2, rkc1, rkc2, rkc3, rkc4, rkc5,
          st1, st2, st3, som1, som2, som3 );

    fH = OpenFile( file, VAL_WRITE_ONLY, VAL_TRUNCATE, VAL_ASCII );
    WriteLine( fH, nodeid, -1 );
    WriteLine( fH, ldev, -1 );
    WriteLine( fH, rdev, -1 );
    WriteLine( fH, data, -1 );
    CloseFile( fH );
  }
}

int CVICALLBACK manual (int panel, int control, int event,
    void *callbackData, int eventData1, int eventData2)
{
  int manual;

  switch (event)
  {
    case EVENT_COMMIT:
      GetCtrlVal( panel, MAINPANEL_BIN_FAST_MANUAL, &manual );
      SetCtrlAttribute( panel, MAINPANEL_NUM_FAST_THRESH, ATTR_DIMMED, manual );
      break;
  }
  return 0;
}
```

### B.2.2 Stereo Library

**stereo.h**

```
#ifndef STEREO_H
#define STEREO_H

typedef struct { double x, y; } dxy;
typedef struct { double x, y, z; } dxyz;
```

```
dxyz stereo_make_dxyz( double * P );
double ** stereo_rodrigues( double theta, double phi, double psi );
double ** stereo_essential( double theta, double phi, double psi, double * T );
void stereo_normalize( int x, int y, double fc1, double fc2, double cc1,
                       double cc2, double kc1, double kc2, double kc3,
                       double kc4, double kc5, dxy * Cn );
dxyz stereo_triangulate( dxy CLn, dxy CRn, double ** R, double * T );

#endif
```

**stereo.c**

```
#include <ansi_c.h>
#include <math.h>
#include "stereo.h"

dxyz stereo_make_dxyz( double * P )
{
  dxyz p;

  p.x = P[ 0 ];
  p.y = P[ 1 ];
  p.z = P[ 2 ];

  return p;
}

double ** stereo_rodrigues( double theta, double phi, double psi )
{
  int i;
  double ** R;

  R = ( double ** )malloc( 3 * sizeof( double * ) );
  for( i = 0; i < 3; i++ )
    R[ i ] = ( double * )malloc( 3 * sizeof( double ) );

  R[ 0 ][ 0 ] = cos( phi ) * cos( psi );
  R[ 0 ][ 1 ] = sin( theta ) * sin( phi ) * cos( psi )
                - cos( theta ) * sin( psi );
```

```
  R[ 0 ][ 2 ] = cos( theta ) * sin( phi ) * cos( psi )
              + sin( theta ) * sin( psi );
  R[ 1 ][ 0 ] = cos( phi ) * sin( psi );
  R[ 1 ][ 1 ] = sin( theta ) * sin( phi ) * sin( psi )
              + cos( theta ) * cos( psi );
  R[ 1 ][ 2 ] = cos( theta ) * sin( phi ) * sin( psi )
              - sin( theta ) * cos( psi );
  R[ 2 ][ 0 ] = -sin( phi );
  R[ 2 ][ 1 ] = sin( theta ) * cos( phi );
  R[ 2 ][ 2 ] = cos( theta ) * cos( phi );

  return R;
}

double ** stereo_essential( double theta, double phi, double psi, double * T )
{
  int i, j;
  double ** R, ** E;
  double Tx[ 3 ][ 3 ];

  R = stereo_rodrigues( theta, phi, psi );

  E = ( double ** )malloc( 3 * sizeof( double * ) );
  for( i = 0; i < 3; i++ )
    E[ i ] = ( double * )malloc( 3 * sizeof( double ) );

  for( i = 0; i < 3; i++ )
  {
    Tx[ i ][ i ] = 0.0;
    Tx[ i ][ ( i + 1 ) % 3 ] = -T[ ( i + 2 ) % 3 ];
    Tx[ i ][ ( i + 2 ) % 3 ] = T[ ( i + 1 ) % 3 ];
  }

  for( i = 0; i < 3; i++ )
  for( j = 0; j < 3; j++ )
  {
    E[ i ][ j ] = R[ i ][ 0 ] * Tx[ 0 ][ j ] + R[ i ][ 1 ] * Tx[ 1 ][ j ]
                + R[ i ][ 2 ] * Tx[ 2 ][ j ];
  }
```

```
  return E;
}

void stereo_normalize( int x, int y, double fc1, double fc2, double cc1,
                       double cc2, double kc1, double kc2, double kc3,
                       double kc4, double kc5, dxy * Cn )
{
  int i;
  double r2, k_radial, xin, yin;

  /* focal length and principal point */

  Cn->x = xin = ( ( double )x - cc1 ) / fc1;
  Cn->y = yin = ( ( double )y - cc2 ) / fc2;

  /* radial/tangential distortion */

  if( kc1 != 0 || kc2 != 0 || kc3 != 0 || kc4 != 0 || kc5 != 0 )
    for( i = 0; i < 20; i++ )
    {
      r2 = Cn->x * Cn->x + Cn->y * Cn->y;
      k_radial = 1 + kc1 * r2 + kc2 * r2 * r2 + kc5 * r2 * r2 * r2;
      Cn->x = ( xin - ( 2 * kc3 * Cn->x * Cn->y + kc4
              * ( r2 + 2 * Cn->x * Cn->x ) ) ) / k_radial;
      Cn->y = ( yin - ( 2 * kc4 * Cn->x * Cn->y + kc3
              * ( r2 + 2 * Cn->y * Cn->y ) ) ) / k_radial;
    }
}

dxyz stereo_triangulate( dxy CLn, dxy CRn, double ** R, double * T )
{
  int i;
  double xt[ 3 ], xtt[ 3 ], u[ 3 ], XL[ 3 ];
  double DD, Zt, Ztt, n_xt2, n_xtt2;
  double dot_uT, dot_xttT, dot_xttu;

  xt[ 0 ] = CLn.x;
  xt[ 1 ] = CLn.y;
```

```
  xtt[ 0 ] = CRn.x;
  xtt[ 1 ] = CRn.y;
  xt[ 2 ] = xtt[ 2 ] = 1.0;
  dot_uT = dot_xttT = dot_xttu = n_xt2 = n_xtt2 = 0.0;

    for( i = 0; i < 3; i++ )
    {
      u[ i ] = R[ i ][ 0 ] * xt[ 0 ] + R[ i ][ 1 ] * xt[ 1 ]
               + R[ i ][ 2 ] * xt[ 2 ];
      n_xt2 += xt[ i ] * xt[ i ];
      n_xtt2 += xtt[ i ] * xtt[ i ];
    }

    for( i = 0; i < 3; i++ )
    {
      dot_uT += u[ i ] * T[ i ];
      dot_xttT += xtt[ i ] * T[ i ];
      dot_xttu += u[ i ] * xtt[ i ];
    }

    DD = n_xt2 * n_xtt2 - dot_xttu * dot_xttu;
    Zt = ( dot_xttu * dot_xttT - n_xtt2 * dot_uT ) / DD;
    Ztt = ( n_xt2 * dot_xttT - dot_uT * dot_xttu ) / DD;

    for( i = 0; i < 3; i++ )
      XL[ i ] = 0.5 * ( ( xt[ i ] * Zt )
                + ( ( R[ 0 ][ i ] * xtt[ 0 ] + R[ 1 ][ i ] * xtt[ 1 ]
                + R[ 2 ][ i ] * xtt[ 2 ] ) * Ztt - T[ i ] ) );

  return stereo_make_dxyz( XL );
}
```

# Bibliography


[1] M. Akdere, U. Cetintemel, D. Crispell, J. Jannotti, J. Mao, and G. Taubin, "Data-Centric Visual Sensor Networks for 3D Sensing," *Proc. 2nd Intl. Conf. on Geosensor Networks*, 2006.

[2] J. Jannotti and J. Mao, "Distributed Calibration of Smart Cameras," *Proc. Intl. Wkshp. on Distributed Smart Cameras*, pp. 55–61, 2006.

[3] K. Obraczka, R. Manduchi, and J. J. Garcia-Luna-Aveces, "Managing the Information Flow in Visual Sensor Networks," *Proc. 5th Intl. Symp. on Wireless Personal Multimedia Communications*, vol. 3, pp. 1177–1181, 2002.

[4] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker, "GHT: A Geographic Hash Table for Data-Centric Storage," *Proc. 1st ACM Intl. Wkshp. on Wireless Sensor Networks and Applications*, 2002.

[5] A. Rao, S. Ratnasamy, C. Papadimitriou, S. Shenker, and I. Stoica, "Geographic Routing Without Location Information," *Proc. 9th Intl. Conf. on Mobile Computing and Networking*, pp. 96–108, 2003.

[6] D. Devarajan and R. J. Radke, "Distributed Metric Calibration of Large Camera Networks," *Proc. 1st Wkshp. on Broadband Advanced Sensor Networks*, 2004.

[7] W. E. Mantzel, H. Choi, and R. G. Baraniuk, "Distributed Camera Network Localization," *Proc. 38th Asilomar Conf. on Signals, Systems and Computers*, 2004.

[8] S. Funiak, C. Guestrin, M. Paskin, and R. Sukthankar, "Distributed Localization of Networked Cameras," *Proc. 5th Intl. Conf. on Information Processing in Sensor Networks*, pp. 34–42, 2006.

[9] C. Beall and H. Qi, "Distributed Self-Deployment in Visual Sensor Networks," *Proc. Intl. Conf. on Control, Automation, Robotics and Vision*, pp. 1–6, 2006.

[10] C. J. Taylor and B. Shirmohammadi, "Self Localizing Smart Camera Networks and their Applications to 3D Modeling," *Proc. Intl. Wkshp. on Distributed Smart Cameras*, pp. 46–50, 2006.

[11] P. J. Besl, "Active, Optical Range Imaging Sensors," *Machine Vision and Applications*, vol. 1, no. 2, pp. 127–152, 1988.

[12] J. Salvi, C. Matabosch, D. Fofi, and J. Forest, "A Review of Recent Range Image Registration Methods with Accuracy Evaluation," *Image and Vision Computing*, vol. 25, no. 5, pp. 578–596, 2007.

[13] C.-S. Chen, Y.-P. Hung, and J.-B. Cheng, "RANSAC-Based DARCES: A New Approach to Fast Automatic Registration of Partially Overlapping Range Images," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 21, no. 11, pp. 1229–1234, 1999.

[14] P. J. Besl and N. D. McKay, "A Method for Registration of 3-D Shapes," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 14, no. 2, pp. 239–256, 1992.

[15] D. Chetverikov, D. Svirko, D. Stepanov, and P. Krsek, "The Trimmed Iterative Closest Point Algorithm," *Proc. Intl. Conf. on Pattern Recognition*, pp. 545–548, 2002.

[16] C. Harris and M. Stephens, "A Combined Corner and Edge Detector," *Proc. Alvey Vision Conference*, pp. 147–151, 1988.

[17] S. M. Smith and J. M. Brady, "SUSAN - A New Approach to Low Level Image Processing," *Intl. Journal of Computer Vision*, vol. 23, no. 1, pp. 45–78, 1997.

[18] A. Baumberg, "Reliable Feature Matching Across Widely Separated Views," *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pp. 1774–1781, 2000.

[19] K. Mikolajczyk and C. Schmid, "Scale and Affine Invariant Interest Point Detectors," *Intl. Journal of Computer Vision*, vol. 60, no. 1, pp. 63–86, 2004.

[20] D. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *Intl. Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[21] K. Mikolajczyk and C. Schmid, "A Performance Evaluation of Local Descriptors," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 10, no. 27, pp. 1615–1630, 2005.

[22] E. Rosten and T. Drummond, "Fusing Points and Lines for High Performance Tracking," *Proc. 10th IEEE Intl. Conf. on Computer Vision*, pp. 1508–1511, 2005.

[23] E. Rosten and T. Drummond, "Machine Learning for High-Speed Corner Detection," *Proc. 9th European Conf. on Computer Vision*, pp. 430–443, 2006.

[24] H. Bay, T. Tuytelaars, and L. Van Gool, "SURF: Speeded Up Robust Features," *Proc. 9th European Conf. on Computer Vision*, pp. 404–417, 2006.

[25] P. Moreels and P. Perona, "Evaluation of Features Detectors and Descriptors Based on 3D Objects," *Proc. 10th IEEE Intl. Conf. on Computer Vision*, pp. 800–807, 2005.

[26] E. Vincent and R. Laganière, "Matching with Epipolar Gradient Features and Edge Transfer," *Proc. Intl. Conf. on Image Processin*, pp. 277–280, 2003.

[27] E. Vincent and R. Laganière, "Models from Image Triplets Using Epipolar Gradient Features," *Image and Vision Computing*, vol. 25, no. 11, pp. 1699–1708, 2007.

[28] D. Scharstein and R. Szeliski, "A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms," *Intl. Journal of Computer Vision*, vol. 47, no. 1, pp. 7–42, 2002.

[29] J. J. Koenderink and A. J. van Doorn, "Geometry of Binocular Vision and a Model for Stereopsis," *Biological Cybernetics*, vol. 21, pp. 29–35, 1976.

[30] R. Y. Tsai, "An Efficient and Accurate Camera Calibration Technique for 3D Machine Vision," *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pp. 364–374, 1986.

[31] R. Y. Tsai, "A Versatile Camera Calibration Technique for High-Accuracy 3D Machine Vision Metrology Using Off-the-Shelf TV Cameras and Lenses," *IEEE Journal of Robotics and Automation*, vol. 3, no. 4, pp. 323–344, 1987.

[32] Z. Zhang, "A Flexible New Technique for Camera Calibration," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 22, no. 11, pp. 1330–1334, 2000.

[33] H. C. Longuet-Higgins, "A Computer Algorithm for Reconstructing a Scene from Two Projections," *Nature*, vol. 293, pp. 133–135, 1981.

[34] E. W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusing Computations," *Information Processing Letters*, vol. 11, no. 1, pp. 1–4, 1980.

[35] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed Deadlock Detection," *ACM Trans. on Computer Systems*, vol. 1, no. 2, pp. 144–156, 1983.

[36] H. A. L. van Dijck, F. van der Heijden, and M. J. Korsten, "Model Based Object Recognition Using Stereo Vision and Geometric Hashing," *Proc. 2nd Ann. Conf. for the Advanced School for Computing and Imaging*, pp. 253–358, 1996.

[37] Y. Sumi and F. Tomita, "3D Object Recognition Using Segment-Based Stereo Vision," *Proc. Asian Conf. on Computer Vision*, pp. 249–256, 1998.

[38] E. Cuevas, D. Zaldivar, and R. Rojas, "Stereo Tracking," *Technical Report B 05-17*, Freie Universitt Berlin, Fachbereich Mathematik und Informatik, 2005.

[39] S. M. Khan, O. Javed, and M. Shah, "Tracking in Uncalibrated Cameras with Overlapping Field of View," *Proc. Intl. Wkshp. on Performance Evaluation of Tracking and Surveillance*, 2001.

[40] Y. Li, A. Hilton, and J. Illingworth, "A Relaxation Algorithm for Real-Time Multiple View 3D-Tracking," *Image and Vision Computing*, vol. 20, no. 12, pp. 841–859, 2002.

[41] E. D. Cheng and M. Piccardi, "Matching of Objects Moving Across Disjoint Cameras," *Proc. Intl. Conf. on Image Processing*, pp. 1769–1772, 2006.

[42] E. G. Rieffel, A. Girgensohn, D. Kimber, T. Chen, and Q. Liu, "Geometric Tools for Multicamera Surveillance Systems," *Proc. Intl. Conf. on Distributed Smart Cameras*, pp. 132–139, 2007.

[43] F. M. S. Ramos and F. M. Patricio, "Application of Distributed Platforms in a Video Surveillance System," *Real-Time Imaging*, vol. 7, no. 5, pp. 447–455, 2001.

[44] M. Quaritsch, M. Kreuzthaler, B. Rinner, and B. Strobl, "Decentralized Object Tracking in a Network of Embedded Smart Cameras," *Proc. Intl. Wkshp. on Distributed Smart Cameras*, pp. 99–105, 2006.

[45] S. Fleck, F. Busch, P. Biber, and W. Strasser, "3D Surveillance A Distributed Network of Smart Cameras for Real-Time Tracking and its Visualization in 3D," *Proc. Conf. on Computer Vision and Pattern Recognition*, pp. 118–118, 2006.

[46] O. Faugeras, N. Ayache, and Z. Zhang, "A Preliminary Investigation of the Problem of Determining Ego- and Object Motions from Stereo," *Proc. Intl. Conf. on Pattern Recognition*, pp. 242–246, 1988.

[47] J.-Y. Shieh, H. Zhuang, and R. Sudhakar, "Motion Estimation From A Sequence Of Stereo Images: A Direct Method," *IEEE Trans. on Systems, Man and Cybernetics*, vol. 24, no. 7, pp. 1044–1053, 1994.

[48] M. Ringer and J. Lasenby, "Modelling and Tracking Articulated Motion from Multiple Camera Views," *Proc. British Machine Vision Conf.*, 2000.

[49] L. Snidaro, C. Piciarelli, and G. L. Foresti, "Activity Analysis for Video Security Systems," *Proc. IEEE Intl. Conf. on Image Processing*, pp. 1753–1756, 2006.

[50] C. Wu and H. Aghajan, "Gesture Analysis in Multi-Camera Networks," *Proc. Intl. Wkshp. on Distributed Smart Cameras*, pp. 110–115, 2006.

[51] C. Wu and H. Aghajan, "Collaborative Face Orientation Detection in Wireless Image Sensor Networks," *Proc. Intl. Wkshp. on Distributed Smart Cameras*, pp. 115–120, 2006.

[52] J. Matas, O. Chum, M. Urban, and T. Pajdla, "Robust Wide-Baseline Stereo from Maximally Stable Extremal Regions," *Image and Vision Computing*, vol. 22, no. 10, pp. 761–767, 2004.

[53] P. Firoozfam and S. Negahdaripour, "Theoretical Accuracy Analysis of N-Ocular Vision Systems for Scene Reconstruction, Motion Estimation, and Positioning," *Proc. 2nd Intl. Symp. on 3D Data Processing, Visualization, and Transmission*, pp. 888–895, 2004.

[54] S. Soro and W. Heinzelman, "Camera Selection in Visual Sensor Networks," *Proc. IEEE Conf. on Advanced Video and Signal Based Surveillance*, pp. 81–86, 2007.

[55] O. Younis and S. Fahmy, "A Scalable Framework for Distributed Time Synchronization in Multi-hop Sensor Networks," *Proc. 2nd IEEE Communications Society Conf. on Sensor and Ad Hoc Communications and Networks*, pp. 13–23, 2005.

[56] Q. Li and D. Rus, "Global Clock Synchronization in Sensor Networks," *IEEE Trans. on Computers*, vol. 55, no. 2, pp. 214–226, 2006.

[57] M. Raynal, *Distributed Algorithms and Protocols*, John Wiley & Sons, 1988.

[58] O. Faugeras, *Three-Dimensional Computer Vision: A Geometric Viewpoint*, The MIT Press, 1993.

[59] Y. Ma, S. Soatto, J. Košecká, S. S. Sastry, *An Invitation to 3-D Vision: From Images to Geometric Models*. Springer-Verlag, 2004.

[60] E. Trucco and A. Verri, *Introductory Techniques for 3-D Computer Vision*, Prentice Hall, 1998.

[61] Z. Zhang and O. Faugeras, *3D Dynamic Scene Analysis: A Stereo Based Approach*, Springer, 1992.

[62] D. Mills, *Network Time Protocol (Version 3): Specification, Implementation and Analysis*, IETF RFC 1305, March 1992; http://www.rfc-editor.org/rfc/rfc1305.txt

[63] A. Jacobs, "An Interest Point Detector and Local Image Descriptor for 3D Rigid Scenes," *IEEE Technical Committee on Digital Libraries Bulletin*, vol. 2, no. 2, http://www.ieee-tcdl.org/Bulletin/v2n2/jacobs/jacobs.html (current 2006).

[64] J. Bouguet, *Camera Calibration Toolbox for Matlab*, http://www.vision.caltech.edu/bouguetj/calib_doc (current April 13, 2007).

[65] Python Software Foundation, *Python 2.5 Documentation*, http://docs.python.org (current September 19, 2006).

[66] SciPy Project, *NumPy Example List*, http://scipy.org/Numpy_Example_List (current February 26, 2008).

# Vita Auctoris

Aaron Mavrinac was born in 1982 in Windsor, Ontario. He graduated from Holy Names High School in 2001. From there he went on to the University of Windsor, where he obtained a B.A.Sc. in Electrical and Computer Engineering in 2005. He is currently a candidate for the Master of Applied Science degree in Electrical and Computer Engineering at the University of Windsor, and plans to graduate in Spring 2008.